Werner Kobitzsch

Mikroprozessoren

Aufbau und Wirkungsweise

Teil 1: Grundlagen

tenverarbeitung
ourg Verlag München Wien

Reihe Datenverarbeitung

Herausgeber:
Klaus Becker-Berke, Rudolf Herschel,
Wolfgang Hilberg, Robert Piloty

Mikroprozessoren
Aufbau und Wirkungsweise

Teil 1: Grundlagen

von Dipl. Ing. **Werner Kobitzsch,**
Standard Elektrik Lorenz AG, Stuttgart

Mit 138 Bildern
und 11 Tabellen

R. Oldenbourg Verlag München Wien 1977

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kobitzsch, Werner
Mikroprozessoren, Aufbau und Wirkungsweise. –
München, Wien : Oldenbourg.
Teil 1. Grundlagen. – 1. Aufl. – 1977.
(Reihe Datenverarbeitung)
ISBN 3-486-20801-2

© 1977 R. Oldenbourg GmbH, München

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, der Funksendung, der Wiedergabe auf photomechanischem oder ähnlichem Wege sowie der Speicherung und Auswertung in Datenverarbeitungsanlagen, bleiben auch bei auszugsweiser Verwertung vorbehalten. Werden mit schriftlicher Einwilligung des Verlages einzelne Vervielfältigungsstücke für gewerbliche Zwecke hergestellt, ist an den Verlag die nach § 54 Abs. 2 Urh.G. zu zahlende Vergütung zu entrichten, über deren Höhe der Verlag Auskunft gibt.

Druck: Hofmann-Druck KG, Augsburg
Bindearbeiten: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-20801-2

Für meine Frau

Inhaltsverzeichnis

Einleitung 9
Liste der verwendeten Symbole und Abkürzungen 11

Kapitel 1
Zahlensysteme: Darstellung und Grundrechenarten 13

1.1 Ableitung einer allgemeinen Zahlendarstellung 13
1.2 Das Dualsystem 14
1.3 Rechnen mit Dualzahlen 18
1.4 Das Oktal- und das Hexadezimalsystem 23
1.5 Das BCD-Zahlensystem 28
1.6 Darstellung von Brüchen in verschiedenen Zahlensystemen (Dual, Oktal, Hexadezimal) 29
1.7 Mehrfach-Genauigkeits- und Gleitkommadarstellung 30

Kapitel 2
Die Baugruppen eines Mikrocomputers 33

2.1 Einführung 33
2.2 Vereinfachtes Blockschaltbild eines Mikrocomputers 33
2.3 Die Zentraleinheit (CPU) 35
2.4 Der Mikrocomputer 68

Kapitel 3
Überblick über Adressierungsarten 79

3.1 Problemstellung 79
3.2 Allgemeine Einteilung der Adressierungsarten 82
3.3 Direkte Adressierung 85
3.4 Direkte Registeradressierungsarten 86
3.5 Direkte Programmzähleradressierungsarten 94
3.6 Indirekte Adressierung 100
3.7 Indirekte Registeradressierungsarten 101
3.8 Indirekte Programmzähleradressierungsarten 106
3.9 Häufige Mikroprozessor-Adressierungsarten 108

Kapitel 4
Befehlsvorrat eines hypothetischen Mikroprozessors 109

4.1 Einteilung des Befehlsvorrats 109
4.2 Arithmetische Befehle 117
4.3 Logische Befehle 119
4.4 Bitkontroll- und Testbefehle 120
4.5 Transferbefehle 122
4.6 Verschiebebefehle 122
4.7 Registerbefehle 124
4.8 Verzweigungs- und Sprungbefehle 127
4.9 Anruf- und Rücksprungbefehle für Unterprogramme 130
4.10 Einfluß der Stackarchitektur auf den Argumententransfer 133
4.11 Ergänzungen zum Befehlsvorrat 137
4.12 Beispiele zum Befehlsvorrat und zu den Adressierungsarten 138

Kapitel 5
Der Mikroprozessor in Verbindung mit Peripheriegeräten 149

5.1 Der Interface-Baustein 149
5.2 Der Datenaustausch zwischen Mikroprozessor und Peripheriegerät 154

Kapitel 6
Softwarehilfsmittel zur Programmerstellung 171

6.1 Vereinbarungen und Definitionen 171
6.2 Der Editor 174
6.3 Der Assembler 179
6.4 Die Debugging Routine 187

Kapitel 7
Mikroprogrammierung: Definition und Grundlagen 191

7.1 Grundlagen 191
7.2 Gegenüberstellung: Herkömmlicher Mikroprozessor und mikroprogrammierbarer Mikroprozessor 192
7.3 Anforderungen an den Anwender 195
7.4 Vorteile und Nachteile der Mikroprogrammierung 197

Literaturverzeichnis 199
Stichwortverzeichnis 200

The advent of the microprocessor as a general-purpose digital electronic block, whose function is determined by programming, is revolutionary.

Gordon E. Moore, Intel

Einleitung

Mikroprozessor und Mikrocomputer sind innerhalb kürzester Zeit zu Schlagworten geworden, denen sich kein Entwickler zu entziehen vermag. Des öfteren wird die Erfindung des Mikroprozessors, die durch Techniken der Hochintegration wesentlich begünstigt wurde, der des Transistors als ebenbürtig gegenübergestellt. Der Mikroprozessor bedeutet für viele Anwender, die bisher mit TTL-Schaltungen kleinerer und mittlerer Integrationsdichte gearbeitet haben, eine große Umstellung. Anwendungen, deren Lösung im Zusammenschalten verschiedenster TTL-Schaltungen bestand, werden, sofern sie mit Mikroprozessoren gelöst werden, aus wenigen hochintegrierten Bauelementen aufgebaut. Die spezifische Problemlösung geschieht über ein vom Anwender zu erstellendes Programm, d.h. mittels Software. Gerade darin besteht das Neue und vielfach Ungewohnte, das die Mikrocomputerlösung mit sich bringt. Der Schaltungsentwickler muß sich mit Begriffen wie Befehlsvorrat, Adressierungsarten und Editor, Assembler sowie Debugging Routine auseinandersetzen. An dieser Stelle will das zweibändige *Lehrbuch* „Mikroprozessoren: Aufbau und Wirkungsweise" ansetzen. In dem vorliegenden Teil 1 werden in 7 Kapiteln die Grundlagen vermittelt, die vom Gesichtspunkt der Hardware und Software für den Entwickler notwendig sind, um sich in das Gebiet der Mikroprozessoren einzuarbeiten. Teil 2 wird typische Mikroprozessoren behandeln und deren Einsatzbereiche aufzeigen. Die in Teil 1 angeführten Kapitel sollen im weiteren von ihrem Inhalt her kurz beschrieben werden.

Kapitel 1 gibt einen Überblick über Zahlensysteme. Dem Leser sei hierin besonders der Abschnitt über Zweierkomplementarithmetik empfohlen.

Kapitel 2 behandelt die Baugruppen eines Mikrocomputers. Nachdem einleitend Definitionen der Begriffe Mikroprozessor und Mikrocomputer gegeben werden, wird schrittweise, ausgehend von einem Halbaddierer, ein Mikrocomputer einschließlich Befehlsvorrat abgeleitet.

Kapitel 3 gibt eine Einführung in die Problematik der Adressierungsarten. Die gebräuchlichsten werden vorgestellt, wobei eine Einteilung in zwei Gruppen „Register-" und „Programmzähleradressierungsarten" vorgeschlagen wird. Ausführliche Beispiele verdeutlichen die jeweils beschriebene Adressierungsart.

Kapitel 4 stellt den Befehlsvorrat eines hypothetischen Mikroprozessors vor. Die definierten Befehle und ihre Anwendung werden aufgezeigt. Die Anwendung bestimmter Adressierungsarten anhand charakteristischer Beispiele schließt Kapitel 4 ab. Es wird dem Leser empfohlen, Kapitel 3 und 4 zusammenhängend zu lesen.

Kapitel 5 beschreibt Datenaustauschverfahren und zeigt den jeweiligen Anwendungsbereich auf.

Kapitel 6 behandelt Softwarehilfsmittel zur Programmerstellung. Im einzelnen werden der Editor, der Assembler und die Debugging Routine vorgestellt.

Kapitel 7 gibt einen Überblick über Mikroprogrammierung. Es wird eine Abgrenzung festprogrammierter-mikroprogrammierter Mikroprozessor durchgeführt.

An dieser Stelle möchte ich all denen danken, die durch ihre Unterstützung, Anregungen und Kritik zum guten Gelingen dieses Buches wesentlich beigetragen haben. Stellvertretend möchte ich Herrn Prof. Dr. Ing. W. Kaiser nennen, der nach einer ersten Durchsicht des Manuskripts die Veröffentlichung in Buchform empfahl. Ganz besonderer Dank gebührt Herrn Prof. Dr. Ing. W. Hilberg, der als verantwortlicher Herausgeber bei der Korrektur des Manuskripts wertvolle Anregungen und Hinweise gab. Der Firma Standard Elektrik Lorenz AG, Stuttgart, sei an dieser Stelle für die Genehmigung zur Buchveröffentlichung gedankt. Dem Verlag R. Oldenbourg gebührt Anerkennung für die bewiesene Geduld und die gelungene Ausstattung des Buches.

Stuttgart, März 1977

Der Verfasser
W. Kobitzsch

Liste der verwendeten Symbole und Abkürzungen

Symbole für boolsche Verknüpfungen

$\wedge$	logisches UND
$\vee$	logisches ODER
$\veebar$	Exklusiv - ODER
$-$	Querstrich symbolisiert eine Negation

Abkürzungen

ACC	accumulator
ALU	arithmetic logic unit
BCD	binary coded decimal
C_n	carry (Übertrag aus der Stelle n−1)
CPU	central processing unit
CU	control unit
DMA	direct memory access
EA	effective address
INC	increment Eingang
k	1 k $\hat{=}$ 1024 = 2^{10} Bit
LIFO	last in first out
LSB	least significant bit
MSB	most significant bit
NOP	no operation
Op-Code	operation code
RAM	random access memory
ROM	read only memory
RePROM	reprogrammable read only memory
PROM	programmable read only memory
PC	program counter

Kapitel 1
Zahlensysteme: Darstellung und Grundrechenarten

1.1 Ableitung einer allgemeinen Zahlendarstellung

Das wohl am meisten bekannte und benutzte Zahlensystem ist das Zehner- oder Dezimalsystem. Dieses System baut auf 10 Zahlen auf und heißt deshalb Dezimalsystem. Bekanntermaßen stehen in diesem System die Ziffern 0 bis 9 zur Zahlendarstellung zur Verfügung.

Da das Dezimalsystem jedoch nicht das einzige gebräuchliche ist, soll im weiteren eine allgemeine Zahlendarstellung für ein System mit der Basis B abgeleitet werden.

Ausgehend von einer Dezimalzahl N_{10}

$$N_{10} = 6 \cdot 10^3 + 5 \cdot 10^2 + 8 \cdot 10^1 + 2 \cdot 10^0 \qquad (1)$$

oder kürzer

$$N_{10} = 6\,5\,8\,2_{10}$$

kann eine allgemeine Zahlendarstellung für ein System der Basis B abgeleitet werden.

$$N_B = A_{n-1} \cdot B^{n-1} + A_{n-2} \cdot B^{n-2} + \ldots + A_0 \cdot B^0 \qquad (2)$$

oder kürzer

$$N_B = A_{n-1}\, A_{n-2} \ldots A_{0_B}.$$

In (2) ist B die Basis des Zahlensystems und $A_{n-1}, A_{n-2}, \ldots, A_0$ sind die Ziffern der Zahl N_B.

Für die Basis und die Ziffern gelten die folgenden Regeln:

Basis: $B > 1$

Ziffern: $A_\nu \in [0, B-1]$ ganz

oder

$A_\nu = (0, 1, 2, \ldots, (B-1)) \qquad \nu = 0, 1, 2, \ldots, n-1$

Aus (2) läßt sich ersehen, daß N_B auch als Summe von Produkten der Form $A_\nu \cdot B^\nu$ dargestellt werden kann. Jeder Ziffer A_ν der Zahl N_B ist ein bestimmter Wert B^ν zugeordnet, das heißt A_ν wird gemäß seinem Stellenwert ν mit B^ν gewichtet.

Der Stellenwert ν beginnt bei der am weitesten rechts stehenden Stelle, dem least significant bit (LSB), mit $\nu = 0$ und erhöht sich jeweils um Eins beim Übergang zur nächsten Stelle. Die Ziffer A_{n-1} wird als das most significant bit (MSB) bezeichnet und mit $B^{\nu} = B^{n-1}$ multipliziert.

Anhand des Beispiels aus dem Dezimalsystem soll das Gesagte veranschaulicht werden.

$$N_{10} = 6582_{10}$$

$$6582_{10} = 6 \cdot 10^3 + 5 \cdot 10^2 + 8 \cdot 10^1 + 2 \cdot 10^0$$

MSB LSB

3	2	1	0	ν = Stellenwert
10^3	10^2	10^1	10^0	Gewichtung

Obige Darstellung wurde für ganze Zahlen abgeleitet, läßt sich jedoch mit Einführung von negativen Stellenwerten und damit von Gewichtungen kleiner 1 auch auf Brüche ausdehnen.

Zum Beispiel:

$$\underbrace{N_B = A_{n-1} \cdot B^{n-1} + A_{n-2} \cdot B^{n-2} + \ldots + A_0 \cdot B^0}_{\text{ganze Zahl}} + \underbrace{A_{-1} \cdot B^{-1} + A_{-2} \cdot B^{-2} + \ldots + A_m \cdot B^m}_{\text{Bruch}}$$

oder kürzer

$$N_B = A_{n-1}\, A_{n-2} \ldots A_0 \cdot A_{-1}\, A_{-2} \ldots A_m.$$

Der Punkt · wird nach dem jeweiligen Zahlensystem, z.B. Dezimalpunkt, Binärpunkt genannt.

1.2 Das Dualsystem

Die meisten digitalen Computer arbeiten mit dem Dualsystem (Basis B = 2), das heißt, alle Berechnungen werden im Dualsystem durchgeführt.

Mit Hilfe der allgemeinen Zahlendarstellung aus Abschnitt 1.1 läßt sich eine Dualzahl sehr einfach ableiten. Da die Basis B = 2 ist, stehen als Ziffern A_{ν} die beiden Werte 0 oder 1 zur Verfügung.

Es ergibt sich somit die Dualzahl N_2 allgemein:

$$N_2 = A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \ldots + A_0 \cdot 2^0 \qquad (3)$$

mit $A_{\nu} = (0, 1) \qquad \nu = 0, 1 \ldots n-1.$

N_2 weist dieselbe Struktur auf wie N_{10}, nur daß die Bewertungsfaktoren B^{ν} jetzt 2^{ν} sind, anstatt wie beim Dezimalsystem 10^{ν}.

1.2.1 Die Dual-Dezimalumwandlung

Da die Dualzahl N_2 eine Summe von Produkten $A_\nu \cdot 2^\nu$ ist, kann die Umwandlung einer Dualzahl in eine entsprechende Dezimalzahl sehr einfach durchgeführt werden.

Gegeben die Dualzahl N_2:

$$N_2 = 1\,0\,1\,0\,1\,1_2$$

oder als Summe dargestellt:

$$N_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0.$$

Nach Addition aller Produktterme $A_\nu \cdot 2^\nu$ mit $A_\nu = 1$ ergibt sich

$$N_{10} = 32 + 8 + 2 + 1 = 43_{10}.$$

1.2.2 Die Dezimal-Dualumwandlung

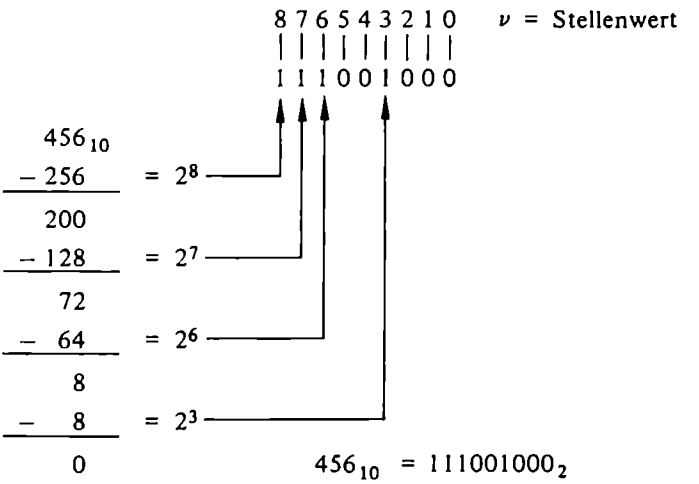
In die Betrachtung sollen zunächst nur positive ganze Zahlen einbezogen werden. Die zur Dezimal-Dualumwandlung herangezogenen Methoden, die Subtraktions- und die Divisionsmethode, sollen anhand eines Beispiels erklärt werden. Anschließend wird für beide Methoden eine allgemeingültige Formel angegeben, nach der eine Umwandlung vom Dezimalsystem in jedes beliebige Zahlensystem durchgeführt werden kann.

Beide Umwandlungsmethoden sind gleichbedeutend und es bleibt dem Anwender überlassen, für welche er sich entscheiden will.

1.2.2.1 Die Subtraktionsmethode

Gegeben sei eine Dezimalzahl N_{10}:

$N_{10} = 456_{10}$

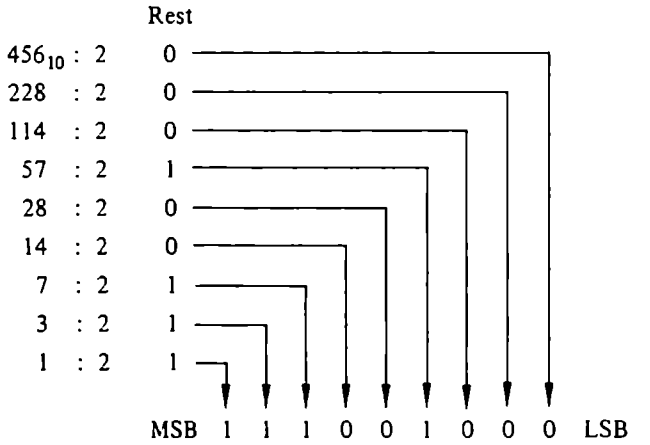


Bei der Subtraktionsmethode wird von der gegebenen Dezimalzahl N_{10} die höchste Zweierpotenz subtrahiert, die in N_{10} enthalten ist. In unserem Beispiel also $256 = 2^8$. Die Ziffer an der Stelle $\nu = 8$ (= MSB) ergibt sich also zu $A_8 = 1$. Es wird jetzt weiter versucht, von der Differenz die nächst niedrigere Zweierpotenz abzuziehen. Ist dies möglich, so ergibt sich als nächste Dualziffer eine 1, andernfalls eine 0 (siehe Beispiel $A_7 = 1$, $A_6 = 1$, $A_5 = 0$, $A_4 = 0$ ...). Diese Subtraktion wird so lange fortgesetzt, bis sich als letzte Differenz 0 ergibt. Wie unser Beispiel zeigt, kann dies bereits bei einer Zweierpotenz 2^ν mit $\nu > 0$ auftreten. Alle weiteren Stellen bis $\nu = 0$ werden mit der Ziffer 0 ergänzt.

1.2.2.2 Die Divisionsmethode

Gegeben sei wiederum die Dezimalzahl N_{10}.

$N_{10} = 456_{10}$



$456_{10} = 111001000_2$

Wie obiges Beispiel zeigt, wird die Dezimalzahl N_{10} fortlaufend durch 2 dividiert. Der dabei entstehende Rest, also entweder 0 oder 1, ergibt die Ziffern der Dualzahl N_2. Es wird zuerst das LSB und zuletzt das MSB berechnet.

1.2.2.3 Die Subtraktionsmethode und die Divisionsmethode in allgemeiner Form

Subtraktionsmethode

$$N_{10} = A_{n-1} \cdot 10^{n-1} + A_{n-2} \cdot 10^{n-2} + \ldots + A_0 \cdot 10^0$$

Umwandlung in das Zahlensystem der Basis C mit dem folgenden Algorithmus:

$$[\{[\underbrace{(A_{n-1} \cdot 10^{n-1} + A_{n-2} \cdot 10^{n-2} + \ldots + A_0 \cdot 10^0) - D_{m-1} \cdot C^{m-1}}_{R_{D_{m-2}}}] - D_{m-2} \cdot C^{m-2}\} - \ldots - D_0 \cdot C^0]$$

mit

$$D_{m-1} \cdot C^{m-1} \leqslant N_{10}$$

und

$$0 < D_{m-1} \leqslant C - 1, \text{ ganz}$$

sowie

$$\left.\begin{aligned} 0 < D_\nu &\leqslant C - 1 \\ D_\nu \cdot C^\nu &\leqslant R_{D_\nu} \end{aligned}\right\} \nu = 0, 1 \ldots m - 2.$$

Aus $D_\nu \cdot C^\nu \leqslant R_{D_\nu}$ folgt:
Ist $C^\nu > R_{D_\nu}$, so muß $D_\nu = 0$ sein (siehe Beispiel 1.2.2.1).

Divisionsmethode

$$N_{10} = A_{n-1} \cdot 10^{n-1} + A_{n-2} \cdot 10^{n-2} + \ldots + A_0 \cdot 10^0$$

oder kürzer

$$N_{10} = A_{n-1} \cdot A_{n-2} \cdot \ldots \cdot A_{0_{10}}.$$

Die Umwandlung in ein Zahlensystem der Basis C erfolgt nach den folgenden Algorithmus, der davon ausgeht, daß eine Zahl als Summe von Produkten der Form $D_\nu \cdot C^\nu$ darstellbar ist mit C als Basis des Systems und D_ν als Ziffer ($D_\nu \in [0, C - 1]$ ganz).

$$N_{10} = N_C = D_{m-1} C^{m-1} + D_{m-2} \cdot C^{m-2} + \ldots + D_1 \cdot C^1 + D_0 \cdot C^0$$

durch Ausklammern:

$$N_{10} = N_C = \underbrace{(D_{m-1} \cdot C^{m-2} + D_{m-2} \cdot C^{m-3} + \ldots D_1)}_{N_{10}^1 \text{ (ganze Zahl)}} C^1 + D_0 \cdot C_0^0 \qquad (1)$$

damit:

$$N_{10} = N_{10}^1 \cdot C^1 + D_0 \cdot C_0^0.$$

Erste Division durch C ergibt:

$$\frac{N_{10}}{C} = N_{10}^1 + \underbrace{\frac{D_0 \cdot C_0^0}{C}}_{\text{Rest } R_{C_0}} \qquad (2)$$

Aus (1) und (2) folgt:

$$\text{Rest } R_{C_0} = \frac{D_0}{C},$$

damit:

$$\Rightarrow D_0 = R_{C_0} \cdot C \text{ das LSB.}$$

N_{10}^1 ist von der Form N_{10}, wobei der höchste Bewertungsfaktor um Eins niedriger ist als bei N_{10}. (→ Ausklammern und Dividieren!).

Die zweite Division wird nun auf N_{10}^1 angewandt. Es ergibt sich:

$$N_{10}^1 = N_{10}^2 \cdot C^1 + D_1 \cdot C^0,$$

nach Division:

$$\frac{N_{10}^1}{C} = N_{10}^2 + \underbrace{\frac{D_1}{C} \cdot C^0}_{\text{Rest } R_{C_1}}$$

$$\text{Rest } R_{C_1} = \frac{D_1}{C}$$

$$\Rightarrow D_1 = R_{C_1} \cdot C$$

Die m-te Division ergibt mit $N_{10}^m = 0$ und $R_{C_{m-1}} \cdot C = D_{m-1}$ das MSB.

1.3 Rechnen mit Dualzahlen

Wir haben nun die Umwandlung vom Dual- zum Dezimalsystem und umgekehrt kennengelernt. Wie eingangs erwähnt, führen die meisten digitalen Computer ihre Rechenoperationen im Dualsystem durch. Aus diesem Grunde sollen in diesem Abschnitt die vier Grundrechnungsarten Addition, Subtraktion, Multiplikation und Division behandelt werden.

1.3.1 Die duale Addition

Gegeben seien zwei Dualzahlen:

	10111_2		23_{10}
+	110_2	+	6_{10}
	$11 \leftarrow$ Übertrag		
	11101_2	=	29_{10}

Zur Bestätigung rechne man die Addition im Dezimalsystem nach.

Obige Addition im Dualsystem zeigt, daß sich ein Übertrag (carry) immer dann ergibt, wenn die Summe einer Spalte größer oder gleich 2 wird.

1.3.2 Die duale Multiplikation

Diese Grundrechenart hängt sehr stark mit der Addition zusammen und soll deshalb im Anschluß an diese behandelt werden.

Gegeben seien zwei Dualzahlen:

10011_2 und 1011_2

$10011_2 \cdot 1011_2$

10011	Stellenwert 0
10011	Stellenwert 1
10011	Stellenwert 3
11111 ← Übertrag	
11010001	

Die duale Multiplikation ist leicht durchzuführen, weil der Multiplikand entsprechend dem Wert des Exponenten der Zweierpotenz nach links verschoben (multipliziert) wird. Somit richtet sich die Anzahl der rechts anzuhängenden Nullen nach dem Stellenwert ν der Zweierpotenz 2^{ν}.

Dies sei mit dem angegebenen Beispiel verdeutlicht.

$10011_2 \cdot 1011_2$

10011	$\nu = 0$	Keine Verschiebung
100110	$\nu = 1$	
0000000	$\nu = 2$	
10011000	$\nu = 3$	Verschiebung um 3 Stellen ⇒ 3 Nullen anhängen

Dieses Verfahren bei der dualen Multiplikation wird bei späteren Multiplikationsprogrammen, die mit Mikroprozessoren durchgeführt werden, noch häufig benötigt werden.

1.3.3 Die duale Subtraktion

Prinzipiell läßt sich eine duale Subtraktion analog zu der im Dezimalsystem durchführen. Häufig wird jedoch in Computern die Zweierkomplementarithmetik angewendet. Diese Arithmetikart läßt sich, wie noch gezeigt wird, auf jedes Zahlensystem anwenden.

1.3.3.1 Die direkte Subtraktion

Gegeben seien die Dualzahlen:

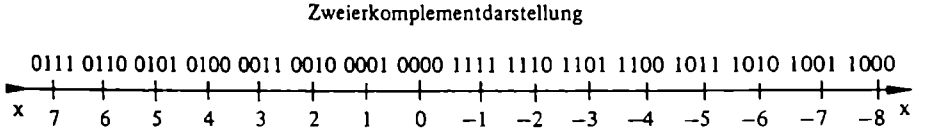
10101_2	21_{10}
$-\ 10011_2$	$-\ 19_{10}$
1 ← Übertrag	→1
00010_2	2_{10}

Ein Übertrag (borrow) ergibt sich, wenn in einer Spalte der Subtrahend größer ist als der Minuend.

1.3.3.2 Die Zweierkomplementarithmetik

Bis jetzt wurden nur ganze Zahlen mit positivem Vorzeichen betrachtet. Die Darstellung von negativen Zahlen kann anhand der Zahlengeraden erfolgen.

Zur Herleitung der Zweierkomplementdarstellung sei ein binäres Zahlensystem mit 4 Stellen (Bits) angenommen, was einem Zahlenumfang von 16 entspricht.



Zahlenstrahl mit Z im Bereich $7 \geq Z \geq -8$

Beim Fortschreiten auf dem Zahlenstrahl bei 0 beginnend nach links ergibt sich die größte positive Zahl (MSB = 0) zu $0111_2 = +7_{10}$. Bewegt man sich nach rechts auf dem Zahlenstrahl, so ergeben sich nacheinander die negativen Zahlen (MSB = 1) -1 bis -8. Das Vorzeichen ist Bestandteil der Zahl. Aus der angegebenen Darstellung läßt sich entnehmen, daß sich bei Addition zweier Zahlen, z.B. $+2_{10}$ und -2_{10}, also 0010_2 und 1110_2, als Summe 0_2 ergibt.

Ob eine Zahl als positiv oder als negativ betrachtet wird, richtet sich nach dem MSB (MSB = 0 positive Zahl, MSB = 1 negative Zahl).

Anhand eines Beispiels aus dem Zahlenbereich $7 \geqslant Z \geqslant -8$ soll gezeigt werden, daß die Zweierkomplementdarstellung ein Weg ist, die duale Subtraktion auf eine duale Addition des Zweierkomplements zurückzuführen.

0011_2		3_{10}
$+\ 1011_2$	-5 in Zweierkomplementdarstellung	$-\ 5_{10}$
$\ \ 11$		
1110_2		$-\ 2_{10}$

1110_2 stellt -2_{10} in Zweierkomplementdarstellung dar.

Da in Mikroprozessoren meistens Dualsysteme mit mehr als 4 Bits vorliegen, so z.B. 8, 16 oder gar 32 Bits, soll die Bildung des Zweierkomplements für eine 8-Bit-Zahl gezeigt werden. In diesem System ist ein Zahlenumfang von 256 realisiert ($-128 \leqslant Z \leqslant 127$).

Gegeben sei eine positive 8-Bit-Zahl

$$01011010_2,$$

gesucht ihr Zweierkomplement.

Der erste Schritt zur Bildung des Zweierkomplements besteht in der Einerkomplementbildung, das heißt, jede Stelle der 8-Bit-Zahl wird invertiert. Das Zweierkomplement ergibt sich nach Addition von + 1 zum Einerkomplement.

Auf das Beispiel angewandt ergibt sich:

(A)	01011010_2	
$(\overline{A})$	10100101_2	Einerkomplement
+ (1)	$+\ 00000001_2$	Addition von + 1
$(\overline{A} + 1)$	10100110_2	Zweierkomplement

Bei Addition der ursprünglichen Zahl A und ihres Zweierkomplements $(\overline{A} + 1)$ *muß* sich nach den Betrachtungen am Zahlenstrahl die Zahl 0 ergeben.

(A)	01011010_2	
$+ (\overline{A} + 1)$	10100110_2	Zweierkomplement
	111111	
→	$1\ 00000000_2$	

dieser Übertrag wird nicht beachtet.

1.3.3.3 Die Subtraktion mit Zweierkomplementarithmetik

Es sei wiederum ein 8-Bit-System zugrunde gelegt.

00010001_2	Minuend	$+ 17_{10}$
-00110101_2	Subtrahend	$- 53_{10}$

Bildung des Zweierkomplements

00110101_2	Subtrahend
11001010_2	Einerkomplement
00000001_2	Addition von + 1
11001011_2	Zweierkomplement

00010001_2		$+ 17_{10}$
$+ 11001011_2$		$- 53_{10}$
11		
$A = 11011100_2$	≙	-36_{10}
−(Zweierkomplement)		-36_{10}

Das Ergebnis bestätigt die Richtigkeit des Verfahrens. A entspricht auf dem Zahlenstrahl der Zahl -36_{10}.

An dieser Stelle sei noch auf eine Besonderheit der Zweierkomplementarithmetik hingewiesen. Bei dem vorgegebenen 8-Bit-System steht uns ein Zahlenbereich Z von − 128 bis + 127 zur Verfügung. Betrachten wir noch einmal eine Subtraktion. (Bei einer Addition tritt dasselbe Phänomen auf.)

11100101_2	Minuend	$- 27_{10}$
-01101110_2	Subtrahend	$- 110_{10}$

Nach Zweierkomplementbildung des Subtrahenden:

$$\begin{array}{rcr} 11100101_2 & & -\ 27_{10} \\ +\ 10010010_2 & & -\ 110_{10} \\ \hline 1\,01110111_2 & \neq & -\ 137_{10} \end{array}$$

↑ Übertrag aus dem Vorzeichenbit

Das Ergebnis einer Subtraktion bzw. Addition in Zweierkomplementarithmetik wird *falsch*, sobald der durch die Anzahl der Bits definierte Zahlenbereich überschritten wird, was man daran erkennt, daß

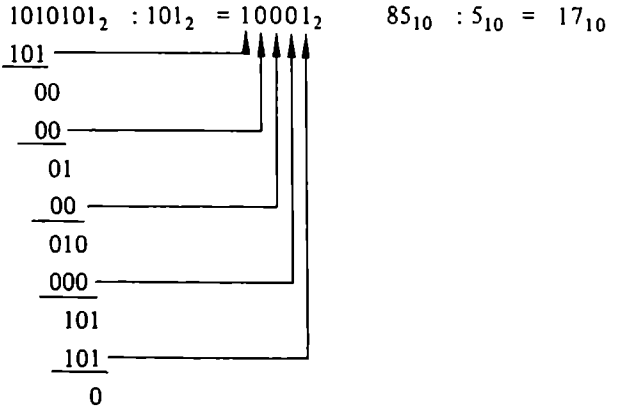
1. das Vorzeichen der Summanden *gleich* ist, aber *ungleich* dem Vorzeichen der Summe (s. obiges Beispiel)
   oder gleichbedeutend damit:
2. bei der Summation entweder ein Übertrag ins Vorzeichenbit oder ein Übertrag aus dem Vorzeichenbit entsteht. (Exklusiv-ODER zwischen Übertrag ins Vorzeichenbit und Übertrag aus Vorzeichenbit).

In vielen Mikroprozessoren wird daher ein Overflow-Flag-Flip-Flop verwendet, dessen Aufgabe es ist, anzuzeigen, wann immer eine Überschreitung des Zahlenbereiches Z, ein „arithmetic overflow", aufgetreten ist. Steht dieses Flag nicht zur Verfügung, so muß ein „arithmetic overflow" durch Software nachgewiesen werden, indem entweder für 1) oder 2) eine Routine geschrieben wird.

1.3.4 Die duale Division

Als letzte der vier Grundrechenarten sei die duale Division vorgestellt. Sie läßt sich, wie das folgende Beispiel zeigt, auf eine wiederholte Subtraktion des Divisors zurückführen.

Division von *ganzen Zahlen*:



1.4 Das Oktal- und das Hexadezimalsystem

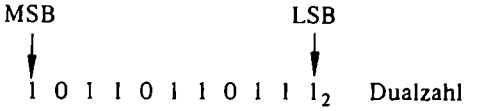
1.4.1 Das Oktalsystem (Basis B = 8)

Um Dezimalzahlen im Dualsystem darzustellen, werden etwa dreimal soviel Dualstellen als Dezimalstellen benötigt. Dies kann vor allem zu Fehlern führen, wenn mehrstellige, also unübersichtliche Dualzahlen abgelesen oder aufgeschrieben werden sollen. Das Oktalsystem, welches jetzt vorgestellt werden soll, schafft hier Abhilfe.

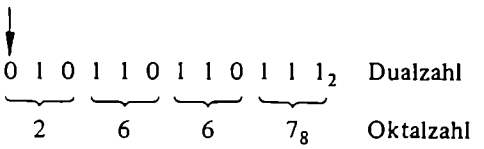
Aus einer Dualzahl ergibt sich die entsprechende Oktalzahl nach folgendem Prinzip:

Jeweils *drei Bit* der Dualzahl werden zu einer Gruppe zusammengefaßt. Dabei wird bei ganzen Zahlen beim LSB begonnen.

Ein Beispiel soll das Prinzip näher erläutern.

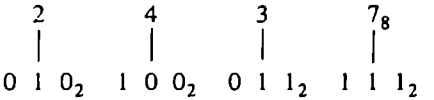


Bildung von Dreiergruppen beginnend beim LSB, d.h. die Dualzahl wird durch $1000_2 = 8_{10}$ dividiert (Divisionsmethode!).



Jede Dreiergruppe wird durch ihr oktales Äquivalent ersetzt. Es ergeben sich Ziffern im Bereich 0 bis 7 (→ Oktalsystem).

Die Rückumwandlung vom Oktal- ins Dualsystem erfolgt durch Substitution der Ziffern der Oktalzahl durch entsprechende dreistellige Dualzahlen (3 Bits, d.h. 8 Zahlen sind darstellbar).

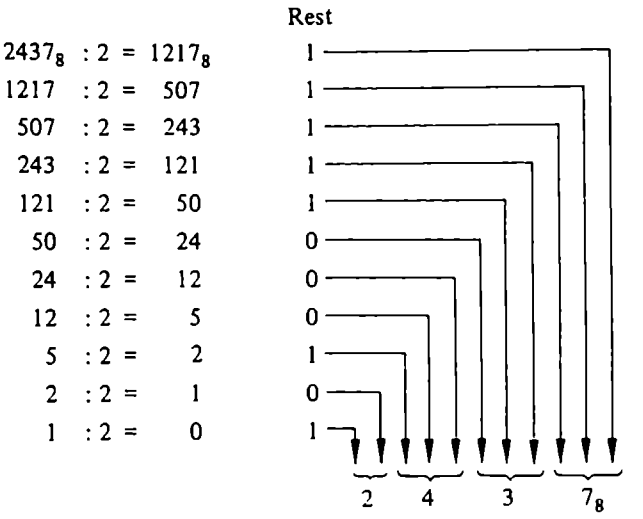


daraus folgt:

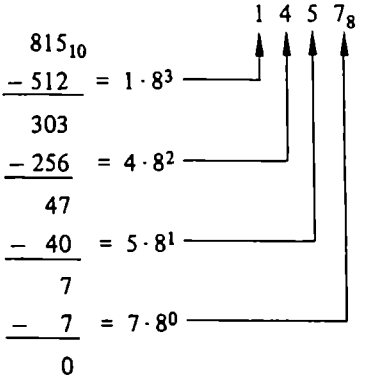
$$2437_8 = 10\ 100\ 011\ 111_2$$

Ganz allgemein erfolgt die Umwandlung von Oktalzahlen in Dual- bzw. Dezimalzahlen und umgekehrt nach den in Abschnitt 1.2 abgeleiteten Methoden. Sie sollen kurz für das Oktalsystem vorgestellt werden.

Oktal-Dualumwandlung nach der Divisionsmethode (1.2.2.2):

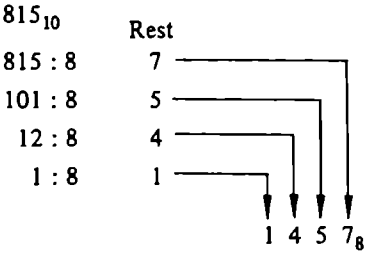


Subtraktionsmethode



$815_{10} = 1457_8$

Divisionsmethode



$815_{10} = 1457_8$

Die Subtraktionsmethode und die Divisionsmethode führen zum selben Ergebnis. Jedoch wird die Subtraktionsmethode seltener angewandt, da aufgrund der zur Verfügung stehenden Ziffern 0 – 7 ein Vielfaches der Bewertungsfaktoren 8^{ν} als Subtrahend in Frage kommen kann. Die Bestimmung des Vielfachen ist zeitraubender als die in der Divisionsmethode angewandte fortlaufende Division durch 8.

Die Oktal-Dezimalumwandlung erfolgt nach der Methode der Addition der Produktterme.

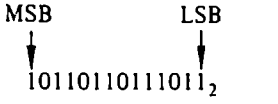
1.4.2 Das Hexadezimalsystem (Basis B = 16)

Tabelle 1 zeigt, daß als Ziffern des Hexadezimalsystems die Zahlen 0 bis 9 und zusätzlich noch die Buchstaben A bis F zur Darstellung der Dezimalzahlen 0 bis 15 herangezogen werden.

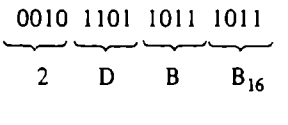
Tabelle 1: Darstellung des Dual-, Oktal-, Hexadezimal- und BCD-Systems

Dezimal	Dual	Oktal	Hexadezimal	BCD
0	000000	0	0	0000 0000
1	000001	1	1	0000 0001
2	000010	2	2	0000 0010
3	000011	3	3	0000 0011
4	000100	4	4	0000 0100
5	000101	5	5	0000 0101
6	000110	6	6	0000 0110
7	000111	7	7	0000 0111
8	001000	10	8	0000 1000
9	001001	11	9	0000 1001
10	001010	12	A	0001 0000
11	001011	13	B	0001 0001
12	001100	14	C	0001 0010
13	001101	15	D	0001 0011
14	001110	16	E	0001 0100
15	001111	17	F	0001 0101
16	010000	20	10	0001 0110
17	010001	21	11	0001 0111
18	010010	22	12	0001 1000
19	010011	23	13	0001 1001
20	010100	24	14	0010 0000
21	010101	25	15	0010 0001
22	010110	26	16	0010 0010
23	010111	27	17	0010 0011
24	011000	30	18	0010 0100
25	011001	31	19	0010 0101
26	011010	32	1A	0010 0110
27	011011	33	1B	0010 0111
28	011100	34	1C	0010 1000
29	011101	35	1D	0010 1001
30	011110	36	1E	0011 0000
31	011111	37	1F	0011 0001
32	100000	40	20	0011 0010

Eine Hexadezimalzahl kann wiederum durch eine entsprechende Bildung von Bitgruppen aus einer Dualzahl abgeleitet werden. Es ist naheliegend, eine Vierergruppe ($10000_2 = 16$) zu wählen, da mit 2^4 Möglichkeiten sich die 16 Ziffern 0 – F darstellen lassen.



Bildung von Vierergruppen



Bei der Umwandlung von Dezimalzahlen in Hexadezimalzahlen bedient man sich einer der Methoden von Abschnitt 1.2.

1.4.3 Rechnen im Oktal- und Hexadezimalsystem

1.4.3.1 Addition

Oktal	*Hexadezimal*
127_8	57_{16}
$+\ 103_8$	$+\ 43_{16}$
232_8	$9A_{16}$

Bei der Addition bedient man sich vorteilhaft der Tabelle 1.

1.4.3.2 Subtraktion

Oktal	*Hexadezimal*
123_8	57_{16}
$-\ 107_8$	$-4A_{16}$
14_8	D_{16}

Ebenso wie bei der Subtraktion im Dualsystem die Zweierkomplementarithmetik angewandt wird, finden im Oktal- und Hexadezimalsystem entsprechend die Achterkomplement- und die Sechzehnerkomplementarithmetik Anwendung.

Oktal		*Hexadezimal*
127_8	Minuend	57_{16}
$-\ 103_8$	Subtrahend	$-\ 43_{16}$

Komplementbildung des Subtrahenden

674_8 ← 7-er Komplement 15-er → BC_{16}
675_8 ← 8-er Komplement 16-er → BD_{16}

Addition des 8-er bzw. 16-er Komplements:

127_8	57_{16}
$+675_8$	$+BD_{16}$
$1\,024_8$	$1\,14_{16}$
↑	↑
wird nicht beachtet	wird nicht beachtet

Beide Ergebnisse stimmen mit den entsprechenden der direkten Subtraktion überein.

1.4.3.3 Multiplikation

Oktal	*Hexadezimal*
$17_8 \cdot 25_8$	$1F_{16} \cdot F2_{16}$
36	1D1
113	3E
473_8	$1D4E_{16}$

Bei der Multiplikation im Oktal- und Hexadezimalsystem wird sehr oft auf die in der Literatur angegebenen Multiplikationstabellen zurückgegriffen [1].

1.4.3.4 Division

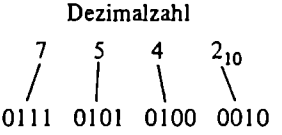
Oktal	*Hexadezimal*
$266_8 : 16_8 = 15_8$	$B6_{16} : E_{16} = D_{16}$
-16	$-B6_{16}$
106	0
-106	
0	

Die Beispiele zeigen, daß das Hexadezimalsystem in seiner Handhabung schwieriger als das Oktalsystem ist, da es ungewohnt ist. Trotzdem findet es neben dem Oktalsystem in Mikroprozessoren Anwendung, und zwar bei der Darstellung des Operationscodes (Op-Code) und von Adressen.

1.5 Das BCD-Zahlensystem

Im BCD (**b**inary **c**oded **d**ecimal)-Zahlensystem wird jeder Ziffer einer Dezimalzahl ihr binäres Äquivalent zugeordnet. Es werden somit die Dualzahlen $0000_2 = 0_{10}$ bis $1001_2 = 9_{10}$ benutzt. Die Nichtverwendung der Dualzahlen 1010_2 bis 1111_2 führt dazu, daß in der BCD-Arithmetik z.B. bei der Addition Korrekturfaktoren hinzugefügt werden müssen (1.5.3).

1.5.1 Die Dezimal-BCD-Umwandlung



Jede Dezimalziffer wird in eine 4-Bit-Dualzahl umgewandelt. Dies erklärt auch, warum dieses Zahlensystem häufig 8, 4, 2, 1-BCD-System genannt wird.

1.5.2 Die BCD-Dualumwandlung

Gegeben sei die BCD-Zahl:

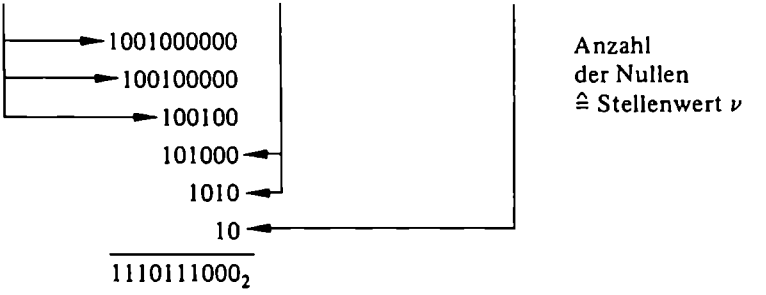
$$1001\ 0101\ 0010.$$

Mit einer anderen Schreibweise ergibt sich:

$$(1001_2) \cdot 10^2 + (0101)_2 \cdot 10^1 + (0010_2) \cdot 10^0$$

Der Bewertungsfaktor 10^ν wird nun ins Dualsystem umgewandelt. Daraus folgt:

$$\underbrace{(1001_2)}_{} \cdot (2^6 + 2^5 + 2^2) + \underbrace{(0101_2)}_{} \cdot (2^3 + 2^1) + \underbrace{(0010_2)}_{} \cdot 2^0$$



Ein Vergleich zwischen BCD-Zahl und Dualzahl bestätigt:

$$952_{10} = 1001\ 0101\ 0010_{BCD} = 1110111000_2$$

1.5.3 Die Addition im BCD-System

BCD	
0001 0101	15_{10}
+ 0000 1001	+ 9_{10}
0001 1110	24_{10}

Existiert nicht im 8, 4, 2, 1 BCD-System

Tritt bei einer Addition eine 4-Bit-Dualzahl auf, die nicht im 8, 4, 2, 1-BCD-System existiert, so *muß* zu dieser Zahl + $6_{10} = 0110_2$ addiert werden. Diesen Vorgang bezeichnet man als Dezimalkorrektur (decimal adjust).

Für unser Beispiel folgt daraus:

0001 1110	
+ 0110	Korrekturfaktor
0010 0100_{BCD}	

Diese Zahl entspricht 24_{10}.

Ein weiterer Fall, der eine Dezimalkorrektur erfordert, liegt vor, sobald sich ein Übertrag aus einer 4-Bit-Gruppe in die nächsthöhere ergibt.

Beispiel:

BCD		
0001 1000		18_{10}
0000 1000		8_{10}
1 ← Übertrag		
0010 0000	≠	26_{10}
+ 0110 ← Korrekturfaktor		
0010 0110_{BCD}	=	26_{10}

1.6 Darstellung von Brüchen in verschiedenen Zahlensystemen (Dual, Oktal, Hexadezimal)

Brüche werden in derselben Form dargestellt wie ganze Zahlen mit dem einen Unterschied, daß der Stellenwert *negativ* ist.

Allgemeine Darstellung eines Bruches:

$$N_B = A_{-1} \cdot B^{-1} + A_{-2} \cdot B^{-2} + \ldots + A_m \cdot B^m .$$

Die Umwandlung eines Bruches des Zahlensystems B_I ins Zahlensystem B_{II} erfolgt nach der *Multiplikationsmethode*.

$$N_{B_I} = A_{-1} \cdot B_{II}^{-1} + A_{-2} \cdot B_{II}^{-2} + \ldots + A_m \cdot B_{II}^{m} .$$

Multiplikation mit B_{II}

$$B_{II} \cdot N_{B_I} = \underbrace{A_{-1}}_{\text{ganze Zahl}} + \underbrace{B_{II}(A_{-2} \cdot B_{II}^{-2} + \ldots + A_m \cdot B_{II}^{m})}_{\text{Bruch}}$$

Auf den verbleibenden Bruch wird wiederum die Multiplikation mit B_{II} angewandt, um auch die anderen Koeffizienten A_{-2}, A_{-3} zu ermitteln.

Achtung: Die Basis B_{II} des neuen Systems *muß* im System B_I dargestellt werden, um die Multiplikationsmethode anwenden zu können.

Beispiel:

$$N_2 = 0.10101_2$$

Umwandlung ins Oktalsystem ($B_{II} = 8$).

Darstellung von B_{II} im Dualsystem

$$B_{II} = 1000_2$$

Anwendung der Multiplikationsmethode

$$0.10101_2 \rightarrow 0$$
$$0.10101_2 \cdot 1000_2 = 101.01 \rightarrow 5$$
$$0.01_2 \cdot 1000_2 = 10.0 \rightarrow 2$$

Diese Methode wird so lange angewandt, bis der verbleibende Bruch identisch Null ist (s. obiges Beispiel).

$$0.10101_2 = 0.52_8$$

Setzt sich eine Zahl aus einer ganzen Zahl und einem echten Bruch zusammen, so kann entweder die ganze Zahl und der echte Bruch getrennt für sich umgewandelt werden oder die entsprechende Zahl wird so normiert, daß entweder eine ganze Zahl oder ein echter Bruch entsteht.

1.7 Mehrfach-Genauigkeits- und Gleitkommadarstellung

1.7.1 **Mehrfach-Genauigkeitsdarstellung** (multi-precision representation)

Vorgegeben sei ein Zahlensystem mit n Bits. Damit lassen sich in Zweierkomplementdarstellung ganze Zahlen im Bereich

$$-2^{n-1} \leqslant Z \leqslant +2^{n-1} - 1$$

darstellen.

Für ein Zahlensystem mit 8 Bits folgt daraus:

$$-2^7 \leqslant Z \leqslant +2^7 - 1$$

oder

$$-128 \leqslant Z \leqslant +127.$$

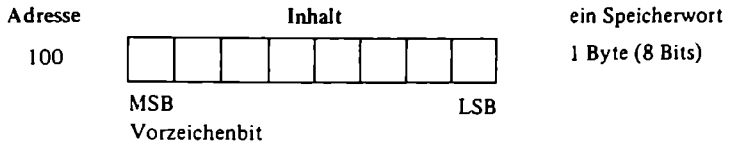
Reicht der Umfang nicht aus, so kann der Zahlenbereich, dadurch daß das m-fache von n Bits zur Genauigkeitsverbesserung herangezogen wird, erweitert werden. In diesem Zusammenhang spricht man von

Doppel-Genauigkeit m = 2
Dreifach-Genauigkeit m = 3 usw.

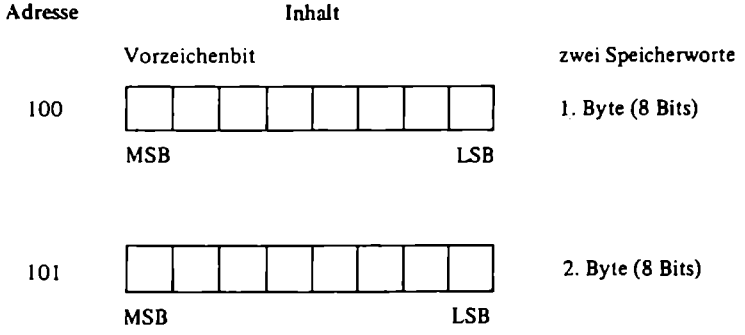
Eine Mehrfach-Genauigkeitszahl wird in einem Mikroprozessorspeicher an mehreren *aufeinanderfolgenden* Speicherplätzen abgespeichert. Der *größere* Bereich zur Darstellung einer Zahl hat zur Folge, daß die Arithmetik, wenn sie im gleichen Mikroprozessorsystem durchgeführt wird, mehr Befehle erfordert und damit langsamer ist, weil z.B. für eine Addition einer doppelgenauen Dualzahl jetzt 2 Additionen ausgeführt werden müssen.

Darstellung einer Einfach-Genauigkeitszahl und einer Doppel-Genauigkeitszahl in einem Speicher:

Einfach-Genauigkeit



Doppel-Genauigkeit:



1.7.2 Die Gleitkommadarstellung (floating point representation)

Eine weitere Methode, um einen größeren Zahlenbereich darstellen zu können, besteht in der Verwendung von Gleitkommazahlen.

Gleitkommazahl

$$Z = M \cdot B^E$$

mit M = Mantisse
B = Basis
E = Exponent

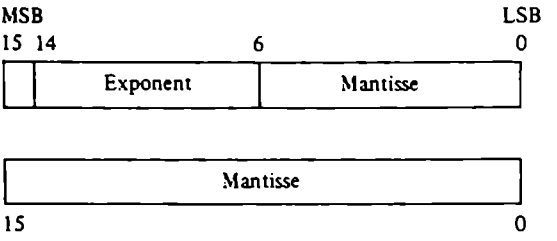
Da die Basis B des Zahlensystems, in dem die Arithmetik durchgeführt werden soll, festliegt, wird die Zahl Z eindeutig durch die Mantisse M und den Exponenten E beschrieben.

Die Gleitkommaarithmetik ist wesentlich aufwendiger als die Festkommaarithmetik, da z.B. bei einer Addition darauf geachtet werden muß, daß die Exponenten der beiden Summanden gleich sind und danach erst die Addition erfolgen kann.

Gleitkommaarithmetik kann entweder in einer gesonderten Hardware erfolgen oder sie wird mittels einer Folge von Festkommaarithmetik- und Logikbefehlen durchgeführt.

Die Gleitkommadarstellung findet überall dort Anwendung, wo die Festkommaarithmetik zu ungenau ist oder die Gefahr eines „arithmetic overflow" gegeben wäre.

Eine 32-Bit-Gleitkommazahl ließe sich wie folgt darstellen:



Für die Mantisse sind 23 Bit, für den Exponenten 8 Bit und für das Vorzeichen 1 Bit vorgesehen.

Die Mantisse wird häufig in der Form dargestellt, daß sie im Bereich

$$\frac{1}{2_{10}} \leqslant M < 1 \text{ liegt.}$$

Für den Exponenten ergibt sich das folgende Schema:

Dezimal	*Dual*
128	11111111
.	.
.	.
.	.
+ 1	10000001
0	10000000
− 1	01111111
.	.
.	.
.	.
− 127	00000000

Kapitel 2
Die Baugruppen eines Mikrocomputers

2.1 Einführung

Wichtig für das Verständnis eines Mikrocomputers ist ein grundlegender Einblick in seinen Aufbau und seine Funktionsweise. An einem vereinfachten Blockschaltbild werden die Baugruppen eines Mikrocomputers vorgestellt. Dabei wird, ausgehend von der arithmetisch-logischen Einheit, anhand deren Befehlsvorrat eine Register-Arithmetikeinheit abgeleitet, die wiederum den Ausgangspunkt bildet für den Mikrocomputer, der aus einfachen Grundbausteinen der TTL-Serie aufgebaut wird. Dieses Vorgehen vom einzelnen Element einer Baugruppe bis zum vollständigen System, dem Mikrocomputer, soll deutlich machen, welche Bedeutung der genauen Kenntnis interner Funktionsabläufe eines Mikrocomputers beigemessen wird. Diese bilden die Grundlage für das Verständnis des Aufbaus und der Wirkungsweise von Mikroprozessoren und den damit möglichen Systemen.

2.2 Vereinfachtes Blockschaltbild eines Mikrocomputers

Das vorherige Kapitel befaßte sich ausführlich mit Zahlensystemen und ihrer Darstellung. Daraus ergeben sich nun die Fragen nach der Verarbeitung dieser Daten, wie die Abarbeitung der dazu notwendigen Befehle (Instruktionen) vor sich geht und wie die Eingabe bzw. Ausgabe der Daten erfolgt. All dies soll im weiteren untersucht werden.

In Bild 2.1 ist ein Mikrocomputer, bestehend aus arithmetisch-logischer Einheit (arithmetic logic unit = ALU), Akkumulator (accumulator = ACC), Steuerwerk, Speicher und Ein-/Ausgaberegister, dargestellt. Aus diesem Blockschaltbild lassen sich die Definitionen der Begriffe „Mikroprozessor" und „Mikrocomputer" ableiten.

Definitionen

Unter einem *Mikroprozessor* wird die Zentraleinheit (central processing unit = CPU) eines Mikrocomputers verstanden. Der Mikroprozessor umfaßt die arithmetisch-logische Einheit (ALU), den Akkumulator und das Steuerwerk. Der Mikroprozessor ist der zentrale Teil, das Herz eines Mikrocomputers. Deshalb wird er oft auch als Zentraleinheit (CPU) bezeichnet.

Unter einem *Mikrocomputer* werden der Mikroprozessor, der Speicher und die Ein-/Ausgaberegister verstanden. Andere Namen sind Mikrorechner, Mikroprozessorsystem.

An Bild 2.1 lassen sich noch einige grundlegende Betrachtungen, die für einen Mikrocomputer wichtig sind, anstellen. Um die zahlreichen Berechnungen durchzuführen, benötigt der Mikroprozessor die arithmetisch-logische Einheit. Die *Steuerung* der mit der ALU möglichen Operationen wird vom Steuerwerk (control unit = CU) vorgenommen. Weiterhin hat das Steuerwerk dafür zu sorgen, daß nach Abarbeitung eines Befehls (Instruktion) ein neuer zur Ausführung bereitgestellt wird. Darüber hinaus sorgt es für die Eingabe und Ausgabe von Daten über die dafür vorgesehenen Ein-/Ausgaberegister. Die Befehle, die die Funktion der ALU und CU festlegen, sind in einem Speicher (memory) abgelegt. Mit diesen Befehlen, dem Programm, ist festgelegt, welche arithmetischen und logischen Funktionen nacheinander durchgeführt werden, woher die Eingabedaten eingelesen und wohin die Ergebnisse ausgegeben werden sollen. Im folgenden werden die verschiedenen Baugruppen in ihrem Aufbau und ihrer Funktion beschrieben.

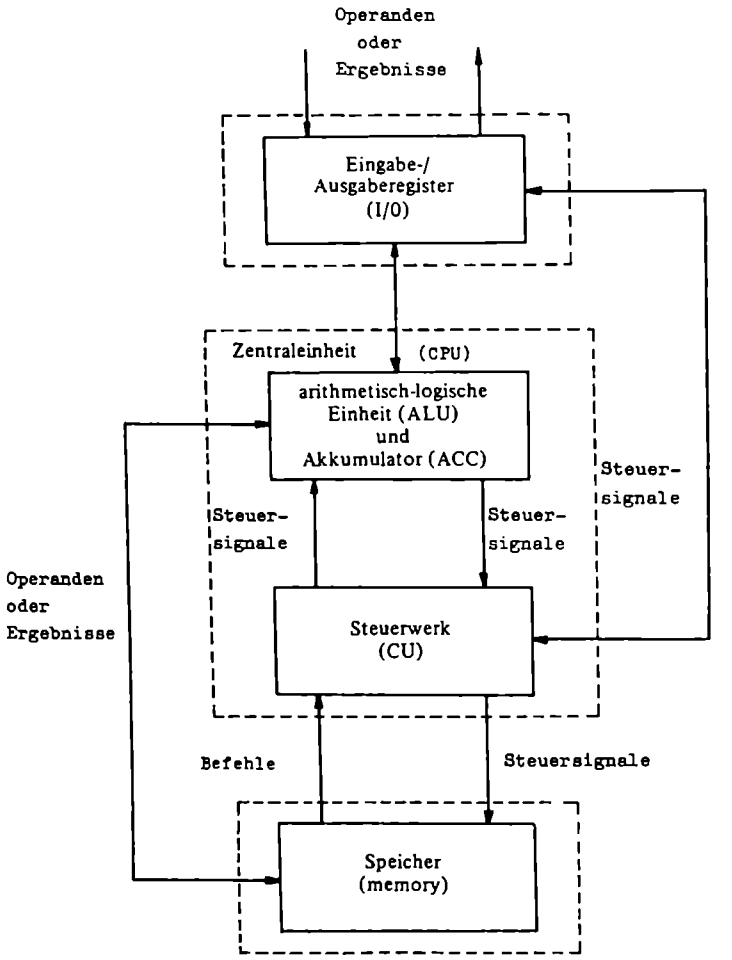


Bild 2.1: Vereinfachtes Blockschaltbild eines Mikrocomputers

2.3 Die Zentraleinheit (CPU)

2.3.1 Die arithmetisch-logische Einheit (ALU)

Aufgrund der allgemeinen Einführung kann für die arithmetisch-logische Einheit ein Befehlsvorrat definiert werden, der die Durchführung arithmetischer und logischer Befehle möglich macht. Die Anforderungen bezüglich der vielfältigen Berechnungen lassen den folgenden Befehlsvorrat notwendig erscheinen.

Befehlsvorrat der ALU:

Addieren mit und ohne Übertrag
Subtrahieren mit und ohne Übertrag
Vermehren um Eins (Increment)
Vermindern um Eins (Decrement)
Komplementieren (Bilden des Einerkomplements)
Negieren (Bilden des Zweierkomplements)
Transferieren
Maskieren (Logisches UND ($\wedge$))
Exklusiv-ODER ($\veebar$)
Inklusiv-ODER ($\vee$)
Verschieben nach links oder rechts

Mit diesem Befehlsvorrat soll der Aufbau einer ALU erklärt werden.

2.3.2 Aufbau der arithmetisch-logischen Einheit

Die weiteren Betrachtungen gehen von einem Halbaddierer aus, der bis zu einer Arithmetikeinheit erweitert wird.

Der Halbaddierer

Die Wahrheitstafel einer Addition zweier 1-Bit-Dualzahlen A und B sieht folgendermaßen aus:

Tabelle 2.1: Wahrheitstafel eines Halbaddierers

A	B	Summe S	Übertrag C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Aus Tabelle 2.1 ergeben sich die Gleichungen für die Summe S und den Übertrag C. Die Gleichung für die Summe S ergibt sich aus der ODER-Verknüpfung aller UND-Verknüpfungen, für die die Summe S logisch „1" wird (Disjunktive Normalform).

$$S = \overline{A} \wedge B \vee A \wedge \overline{B}$$

Dies entspricht der Antivalenz- oder der Exklusiv-ODER ($\veebar$)-Verknüpfung.

Es ergibt sich deshalb:

$$S = A \veebar B.$$

Für C erhält man mit denselben Überlegungen:

$$C = A \wedge B.$$

Aus diesen Gleichungen läßt sich die Schaltkreiskonfiguration eines Halbaddierers ableiten. Bild 2.2 zeigt eine der möglichen Realisierungen.

Der Volladdierer

Bei der Addition von mehrstelligen Dualzahlen muß gewährleistet sein, daß auch der Übertrag aus der vorhergehenden Stelle berücksichtigt wird. Dies wird durch Erweiterung des Halbaddierers auf einen Volladdierer erreicht, indem ein zusätzlicher Eingang für den Übertrag C_n aus der vorhergehenden Stelle hinzugefügt wird.

Tabelle 2.2: Wahrheitstafel eines Volladdierers bei der Addition des n-ten Bits

A_n	B_n	C_n	S_n	C_{n+1}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Die in der Tabelle 2.2 verwendeten Indizes besagen:

- n kennzeichnet die gerade bearbeitete Stelle der Dualzahl.
- C_n Übertrag aus der vorhergehenden Stelle (n – 1), der zur Stelle n hinzuaddiert wird.
- C_{n+1} Übertrag der Stelle n; er wird bei der Addition der (n + 1)-ten Stelle berücksichtigt.

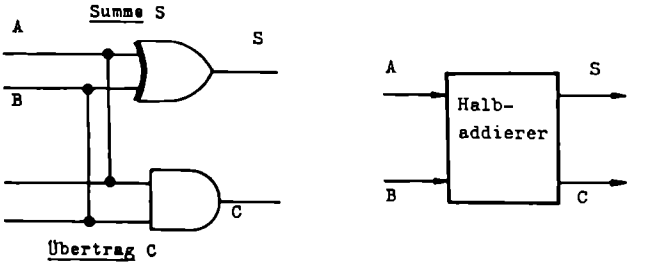


Bild 2.2: Der Halbaddierer

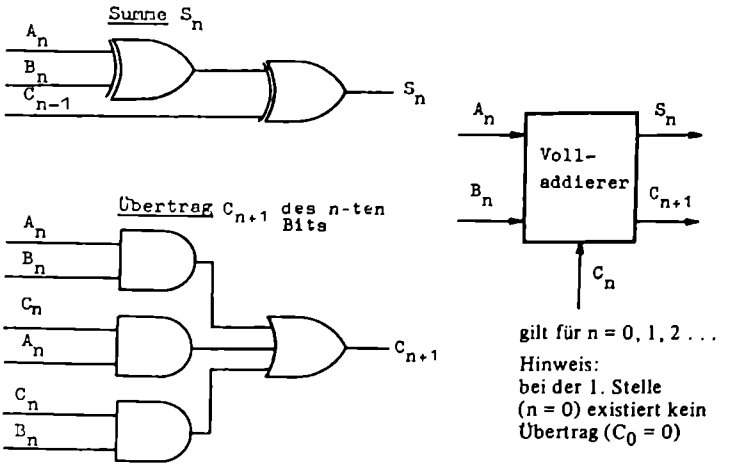


Bild 2.3: Der Volladdierer

Aus Tabelle 2.2 lassen sich analog zum Halbaddierer die Gleichungen für die Summe S_n und den Übertrag C_n eines Volladdierers ableiten.

Gleichung der Summe S_n:

$$S_n = (A_n \wedge \overline{B_n} \vee \overline{A_n} \wedge B_n) \wedge \overline{C_n} \vee (\overline{A_n} \wedge \overline{B_n} \vee A_n \wedge B_n) \wedge C_n$$

oder weiter vereinfacht:

$$S_n = A_n \veebar B_n \veebar C_n \quad \text{bzw.} \quad S_n = \overline{\overline{A_n \veebar B_n \veebar C_n}}. \tag{1}$$

Gleichung des Übertrags C_{n+1}:

$$C_{n+1} = A_n \wedge B_n \vee (A_n \veebar B_n) \wedge C_n. \tag{2}$$

Aus der Wahrheitstafel läßt sich für Gleichung (2) noch eine andere Darstellung ableiten, die besonders in sogenannten Look-Ahead Carry-Schaltkreisen Verwendung findet.

$$C_{n+1} = (A_n \wedge B_n) \vee (A_n \wedge C_n) \vee (B_n \wedge C_n) \tag{3}$$

In Bild 2.3 sind die Schaltkreise angegeben, aus denen ein Volladdierer aufgebaut ist.

2.3.3 Verschiedene Additionstechniken

2.3.3.1 Ripple-Carry-Addierer

Die Geschwindigkeit der hier vorgestellten parallelen Addition zweier n-Bit-Dualzahlen wird dadurch begrenzt, daß ein Übertrag C_1, der sich nach der Addition in der ersten Stelle ergeben kann, sich bis zur n-ten Stelle fortpflanzt. Die korrekte (n + 1)-stellige Summe S ist somit erst vorhanden, wenn alle Überträge C_1 bis C_{n+1} gebildet worden sind.

Zu dem gegebenen Beispiel sei bemerkt, daß es sich hierbei um den ungünstigsten Fall (worst case) handelt.

Einen Addierer, der aus der Serienschaltung von Volladdierern nach Bild 2.3 besteht, nennt man Ripple-Carry-Addierer (Bild 2.4).

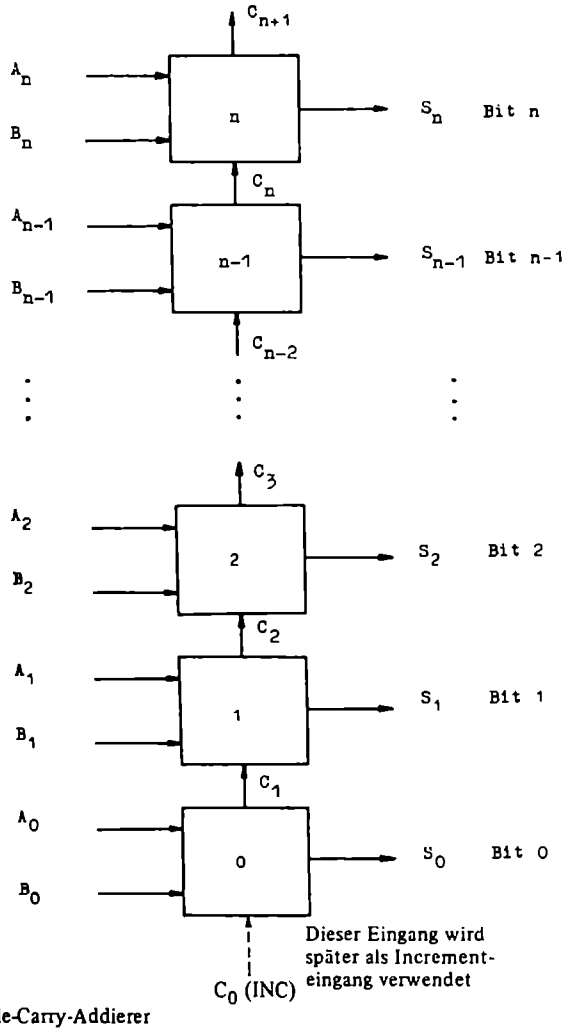


Bild 2.4: Ripple-Carry-Addierer

2.3.3.2 Addierer mit Look-Ahead Carry

Zur Beschleunigung einer Addition bedient man sich des Look-Ahead Carry-Prinzips. Dieses umgeht den oben aufgezeigten Nachteil des Ripple-Carry-Addierers.

Gegeben sei wiederum die Gleichung für den Übertrag C_{n+1} aus 2.3.2

$$C_{n+1} = (A_n \wedge B_n) \vee (A_n \wedge C_n) \vee (B_n \wedge C_n)$$

oder

$$C_{n+1} = (A_n \wedge B_n) \vee [(A_n \vee B_n) \wedge C_n].$$

Diese Gleichung besagt:

Ein Übertrag ergibt sich in zwei Fällen, und zwar

1. wenn A_n *und* B_n logisch wahr sind

oder

2. wenn entweder A_n *oder* B_n logisch wahr sind *und* zugleich C_n logisch wahr ist, das heißt, ein Übertrag aus der vorhergehenden Stelle vorhanden ist.

Mit der Einführung der Symbole G und P soll eine Rekursionsformel für die Addition zweier n-Bit-Dualzahlen abgeleitet werden, dabei steht G für Generierung (generate) eines Übertrags (carry) durch die beiden Stellen A_n und B_n und P für Fortpflanzung (propagate) des in der vorhergehenden Stelle erzeugten Übertrags C_n.

Somit ergibt sich die Schreibweise für C_{n+1}:

$$C_{n+1} = G_n \vee (C_n \wedge P_n) \tag{4}$$

mit

$$G_n = A_n \wedge B_n$$

und

$$P_n = A_n \vee B_n;$$

für C_n gilt analog:

$$C_n = G_{n-1} \vee (C_{n-1} \wedge P_{n-1}). \tag{5}$$

Durch Einsetzen von (5) in (4) ergibt sich die Rekursionsformel für C_{n+1}:

$$C_{n+1} = G_n \vee (P_n \wedge G_{n-1}) \vee (P_n \wedge P_{n-1} \wedge G_{n-2}) \vee \ldots \vee (P_n \wedge P_{n-1} \wedge \ldots \wedge P_0 \wedge \mathrm{INC}). \tag{6}$$

Diese Gleichung zeigt, daß sich der Übertrag C_{n+1} als eine ODER-Verknüpfung darstellen läßt, deren einzelne Terme gleichzeitig gebildet werden können.

Als Beispiel sei eine Addition zweier 4-Bit-Dualzahlen mit Look-Ahead Carry vorgestellt.

Für C_4 ergibt sich nach (6):

$$C_4 = G_3 \vee (P_3 \wedge G_2) \vee (P_3 \wedge P_2 \wedge G_1) \vee (P_3 \wedge P_2 \wedge P_1 \wedge G_0) \vee (P_3 \wedge P_2 \wedge P_1 \wedge P_0 \wedge \mathrm{INC})$$

C_4 ist von der Form:

$$C_4 = G \vee (C \wedge P).$$

Die maximale Verzögerungszeit beträgt für die Bildung von C_4 drei Gatterlaufzeiten. (Das ist die Zeit zur Bildung von $G_n = A_n \wedge B_n$ und $P_n = A_n \vee B_n$, die Zeit zur Bildung der UND-Verknüpfungen sowie der ODER-Gesamtfunktion.)

Im praktischen Fall wird sowohl das Ripple-Carry-Prinzip als auch das Look-Ahead Carry-Prinzip oder eine Mischung aus beiden angewandt. Allein die geforderte Geschwindigkeit und die Anzahl der Schaltkreise ist maßgebend für die verwendete Methode.

In Bild 2.5 ist ein Look-Ahead Carry-Generator (74 S 182) dargestellt.

Bemerkung:

Die Eingänge $\overline{P_0}$ bis $\overline{P_3}$ und $\overline{G_0}$ bis $\overline{G_3}$ sind vorgesehen, um diesen Schaltkreis mit der noch zu besprechenden Akkumulator-Arithmetikeinheit (74 S 281) zusammen schalten zu können.

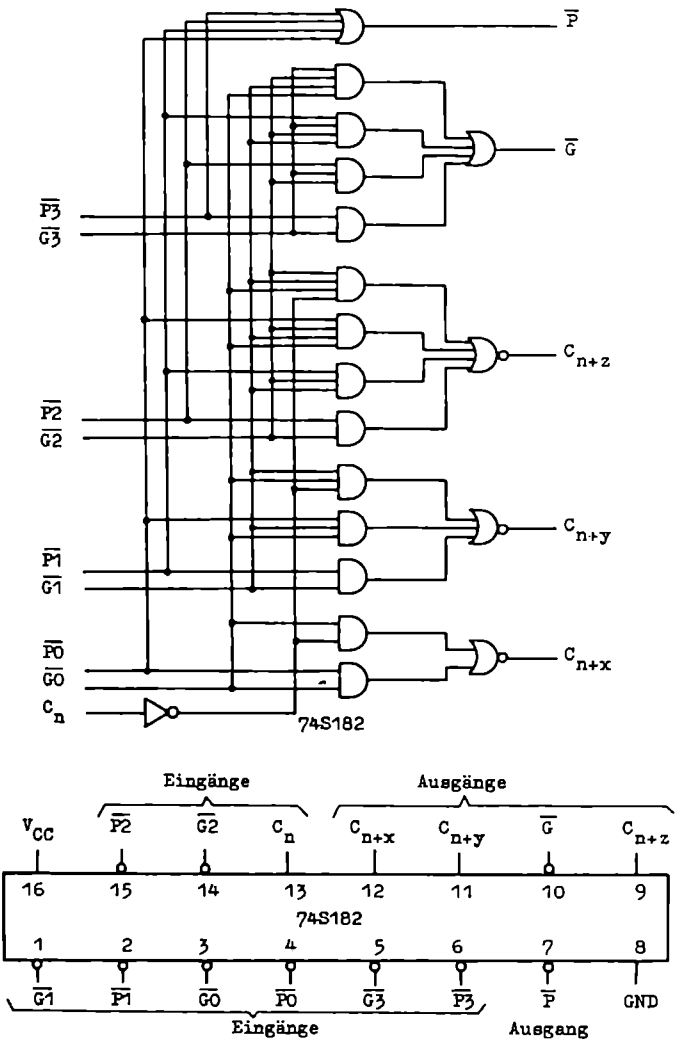


Bild 2.5: Look-Ahead Carry-Generator

2.3.4 Erweiterung des Volladdierers zur Arithmetikeinheit

Aufbauend auf dem Volladdierer kann durch Hinzufügen eines in der Literatur häufig als True-Complement-Schaltkreis (74H87) bezeichneten Bausteins eine Arithmetikeinheit verwirklicht werden. Eine derartige Schaltung und die damit möglichen Funktionen sind in den Bildern 2.6 bis 2.8 dargestellt. Mit der vorgestellten Arithmetikeinheit ist, wie die weiteren Ausführungen zeigen werden, ein Teil der Anforderungen zu erfüllen, die zu Beginn an eine arithmetisch-logische Einheit (ALU) gestellt wurden.

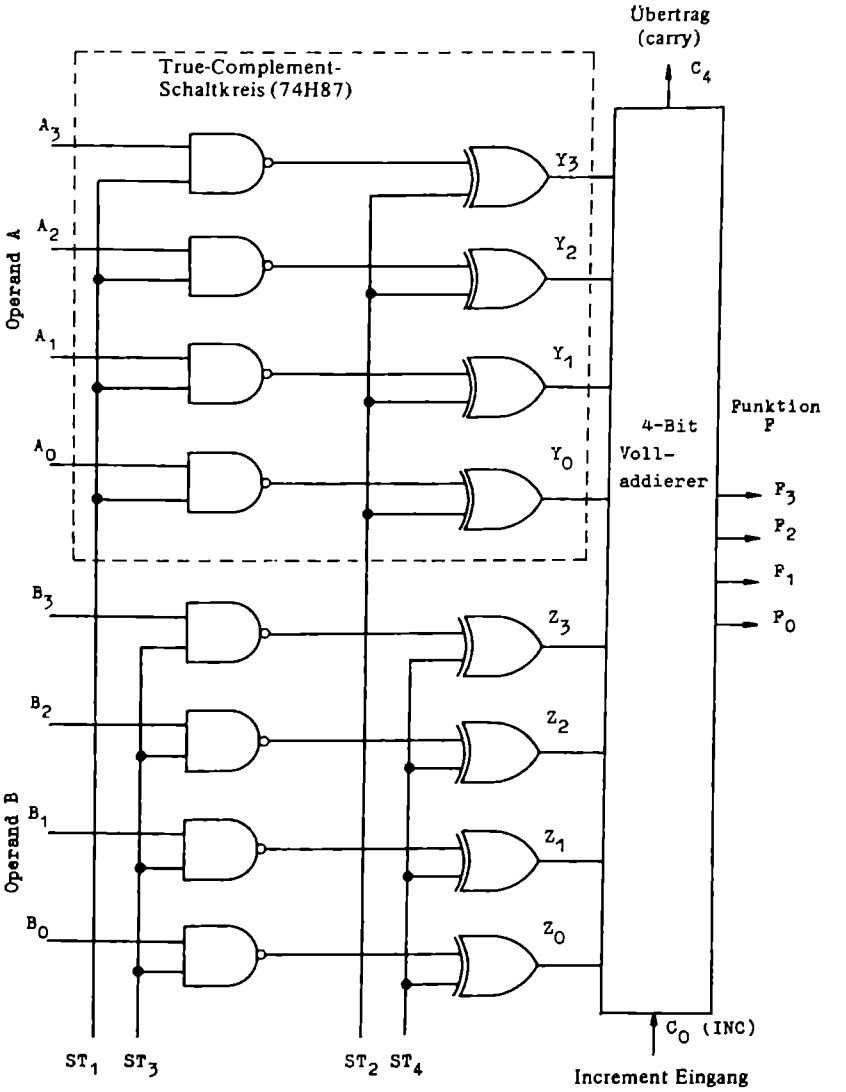


Bild 2.6: 4-Bit-Arithmetikeinheit

Zunächst wird der Aufbau einer 4-Bit-Arithmetikeinheit an Bild 2.6 erklärt. Die Bezeichnung 4-Bit-Arithmetikeinheit besagt, daß die ausführbaren Funktionen mit vierstelligen Operanden möglich sind.

Die Arithmetikeinheit besitzt Eingänge für die zu verarbeitenden Operanden A und B. Mit den Steuereingängen ST1 bis ST4 und C_0 werden in der Arithmetikeinheit Funktionen eingestellt, die mit den Operanden A und B ausgeführt werden können. Bild 2.6 veranschaulicht bezüglich der Operanden den symmetrischen Aufbau der Arithmetikeinheit. Dem 4-Bit-Volladdierer ist für die Eingänge der Operanden A bzw. B jeweils ein 4-Bit-True-Complement-Schaltkreis vorgeschaltet, der über die Steuerleitungspaare ST1 und ST2 sowie ST3 und ST4 die Operanden A bzw. B verschiedenen boolschen Operationen unterwirft. Die Symmetrie im Aufbau der Arithmetikeinheit gestattet es, sich bei der weiteren Betrachtung auf die Steuerleitungen ST1 und ST2 zu beschränken. Beide, ST1 und ST2, führen abhängig von ihren logischen Zuständen dazu, daß der Operand A verändert wird, entsprechend den boolschen Operationen, die für eine NAND- und eine Exklusiv-ODER-Verknüpfung gelten. Um die Entstehung der Wahrheitstafel und damit die Funktionen der 4-Bit-Arithmetikeinheit zu verstehen, sind in Bild 2.7 die Ausgangswerte Y des True-Complement-Schaltkreises abhängig von den Eingangswerten ST1, ST2 und Operand A angegeben. Es sei nochmals bemerkt, daß dasselbe aus Symmetriegründen auch für Operand B gilt. Operand A und Ausgangswert Y sind jeweils 4-Bit-Dualzahlen.

Eingang A ($A_3\ A_2\ A_1\ A_0$)	Steuerleitungen ST1	ST2	Ausgang Y ($Y_3\ Y_2\ Y_1\ Y_0$)	Befehl
$A_3\ A_2\ A_1\ A_0$	0	0	1 1 1 1	Setzen
$A_3\ A_2\ A_1\ A_0$	1	1	$A_3\ A_2\ A_1\ A_0$	Keine Operation (unverändert)
$A_3\ A_2\ A_1\ A_0$	0	1	0 0 0 0	Löschen
$A_3\ A_2\ A_1\ A_0$	1	0	$\overline{A_3}\ \overline{A_2}\ \overline{A_1}\ \overline{A_0}$	Komplementieren (Einerkomplement)

Bild 2.7: Wahrheitstafel des True-Complement-Schaltkreises

Aus der Tabelle (Bild 2.7) ist ersichtlich:

Abhängig von den Steuereingängen ST1 und ST2 erscheint die Eingangsinformation A (Operand A) unverändert oder komplementiert als Ausgangsinformation Y oder es stellt sich unabhängig von der Eingangsinformation A am Ausgang der Wert Y = logisch „0" oder Y = logisch „1" ein. Die Ausgangsinformation $Y = 1111_2$ entspricht – 1 in Zweierkomplementdarstellung. Für jeden dieser vier Fälle haben sich auf dem Gebiet der Datenverarbeitung spezielle Namen eingebürgert. Diese sind ebenfalls im Bild 2.7 für den True-Complement-Schaltkreis angegeben.

Mit der vorliegenden Wahrheitstafel für den True-Complement-Schaltkreis ist es ein leichtes, anhand von Bild 2.7 die Wahrheitstafel für die 4-Bit-Arithmetikeinheit abzuleiten. Sie ergibt sich aus der Addition der Ausgangswerte Y und Z der True-

Complement-Schaltkreise, die im 4-Bit-Volladdierer erfolgt. Die Funktion F (F_3 F_2 F_1 F_0) in der Tabelle in Bild 2.8 macht deutlich, welche Operationen, d.h. Befehle mit der Arithmetikeinheit durchgeführt werden können. Mit anderen Worten stellt die Tabelle dar, welches Ergebnis sich ergibt, und zwar abhängig

1. von Operand A *und* Operand B;
2. *entweder* von Operand A *oder* von Operand B;
3. *unabhängig* von Operand A und Operand B zu logisch „0" oder logisch „1".

Zu der Tabelle in Bild 2.8 sind noch zwei Bemerkungen notwendig.

Erstens:

Die Spalten mit $C_0 = 0$ und $C_0 = 1$ unterscheiden sich nur dadurch, daß bei $C_0 = 1$ das Ergebnis immer um Eins höher ist als bei $C_0 = 0$. Dies erklärt, warum der Eingang C_0 oft treffender als Incrementeingang, abgekürzt INC, bezeichnet wird. Seine Aufgabe ist es, das Ergebnis um Eins zu vermehren (erhöhen engl.: to increment).

Zweitens:

Aus dem Abschnitt über Zweierkomplementarithmetik ist bekannt, daß zu einer positiven Dualzahl (N_2) die negative ($-N_2$) wie folgt gebildet wird:

$$-N_2 = \overline{N_2} + 1.$$

Von diesem Zusammenhang wurde Gebrauch gemacht, um in der Wahrheitstafel der 4-Bit-Arithmetikeinheit arithmetische Operationen (Addition, Subtraktion) in der Form A + B, A − B und B − A darstellen zu können. Dabei ist es jedoch wesentlich, zu wissen, daß die Arithmetikeinheit z.B. bei einer Subtraktion A − B anstelle der Subtraktion − B die Addition von $\overline{B} + 1$ zu A ausführt. Über die Steuerleitungen ST2 bzw. ST4 muß also dafür gesorgt werden, daß Operand A unverändert bleibt und Operand B komplementiert wird. Weiterhin muß am Eingang INC eine logische „1" anliegen. Demnach kann mit ein und derselben Arithmetikeinheit sowohl addiert als auch subtrahiert werden, wodurch abermals der Nutzen der Zweierkomplementarithmetik verdeutlicht wird.

Abschließend sind in einer Zusammenstellung die Befehle angegeben, die mit der Arithmetikeinheit möglich sind. Es sind dies die Befehle:

Addieren,
Subtrahieren,
Vermehren um Eins,
Vermindern um Eins,
Komplementieren (Einerkomplement),
Negieren (Zweierkomplement).

Der vorhandene Befehlsvorrat erfüllt, was die arithmetischen Befehle anbelangt, die gestellten Anforderungen aus Abschnitt 2.3.1 und ermöglicht darüber hinaus noch die Befehle:

Setzen,
Löschen,
Unverändert lassen (no operation = NOP)
(z.B. Transferieren).

				Funktionen F		Befehle	
				INC = 0	INC = 1		
ST_1	ST_2	ST_3	ST_4	Increment-eingang	Increment-eingang	INC = 0	INC = 1
0	0	0	0	- 2	- 1	- 2	Setzen
1	0	0	0	$\bar{A}$ - 1	$\bar{A}$	- A - 2	- A - 1
0	1	0	0	- 1	0	Setzen	Löschen
1	1	0	0	A - 1	A	A - 1	NOP
0	0	1	0	$\bar{B}$ - 1	$\bar{B}$	- B - 2	- B - 1
1	0	1	0	$\bar{A}$ + $\bar{B}$	$\bar{A}$ + $\bar{B}$ + 1	- A - B - 2	- A - B - 1
0	1	1	0	$\bar{B}$	$\bar{B}$ + 1	- B - 1	- B
1	1	1	0	A + $\bar{B}$	A + $\bar{B}$ + 1	A - B - 1	A - B
0	0	0	1	- 1	0	Setzen	Löschen
1	0	0	1	$\bar{A}$	$\bar{A}$ + 1	- A - 1	- A
0	1	0	1	0	1	Löschen	1
1	1	0	1	A	A + 1	NOP	A + 1
0	0	1	1	B - 1	B	B - 1	NOP
1	0	1	1	$\bar{A}$ + B	$\bar{A}$ + B + 1	B - A - 1	B - A
0	1	1	1	B	B + 1	NOP	B + 1
1	1	1	1	A + B	A + B + 1	A + B	A + B + 1

$a_3a_2a_1a_0$ = A $b_3b_2b_1b_0$ = B - A = $\bar{A}$ + 1

Bild 2.8: Wahrheitstafel einer 4-Bit-Arithmetikeinheit

2.3.5 Der Akkumulator (ACC) und die arithmetisch-logische Einheit (ALU)

Bisher wurde nur ein Teil der Anforderungen, die an eine ALU gestellt werden, durch Schaltungsbeispiele abgedeckt, nämlich der der arithmetischen Befehle.

Der Rest der noch zu ermöglichenden Befehle kann einerseits durch zusätzliche Logikschaltungen in Verbindung mit der Arithmetikeinheit verwirklicht werden (UND, Inklusiv-, Exklusiv-ODER). Diese erweitern die Arithmetikeinheit zu einer arithmetisch-logischen Einheit. Andererseits können die noch ausstehenden Verschiebebefehle (links und rechts verschieben) mit Hilfe eines Schieberegisters durchgeführt werden.

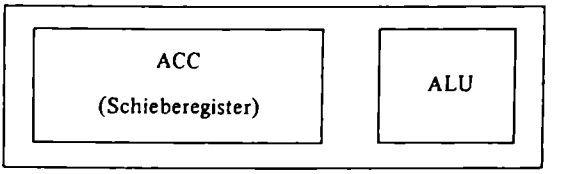
An dieser Stelle sei noch auf zwei Definitionen beim Gebrauch des Begriffs Akkumulator (ACC) hingewiesen, die beide in der Literatur benutzt werden.

Beiden Definitionen ist gemeinsam, daß der Akkumulator zur Speicherung eines zu verarbeitenden Operanden oder eines Ergebnisses dient und daß in ihm Verschiebebefehle möglich sind. Der Name Akkumulator wird aus dem Lateinischen accumulare = anhäufen abgeleitet.

Die Definitionen unterscheiden sich jedoch in bezug auf die arithmetisch-logische Einheit.

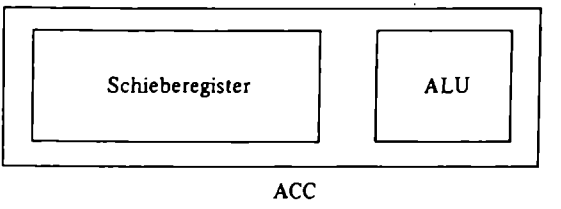
Definition 1:

Als Akkumulator wird ein Register in der Zentraleinheit bezeichnet, das von der arithmetisch-logischen Einheit getrennt ist.



Definition 2:

Als Akkumulator wird ein Register und die arithmetisch-logische Einheit in der Zentraleinheit verstanden.



Die Definition 1 soll im weiteren Verwendung finden, da sie sich später auch ohne große Schwierigkeit auf Zentraleinheiten mit mehreren Akkumulatoren anwenden läßt.

In Bild 2.9 ist das Prinzip der Zusammenschaltung eines Akkumulators mit einer arithmetisch-logischen Einheit (ALU) dargestellt. Diese Konfiguration wird kurz Akkumulator-Arithmetikeinheit genannt. Die Bereitstellung der Operanden, mit denen Befehle ausgeführt werden sollen, erfolgt über Verbindungsleitungen, die mit Operand I und Operand II bezeichnet sind. Für die weitere Betrachtung wird von einem Befehl ausgegangen, der zwei Operanden verknüpft (z.B. Addiere). Operand I komme aus dem Akkumulator. Entsprechend dem Befehl wird dieser mit Operand II in der ALU verknüpft. Das Ergebnis wird von der ALU zum Akkumulator zurückgeführt und dort abgespeichert. Hier steht es als neuer Operand I für weitere Befehle zur Verfügung.

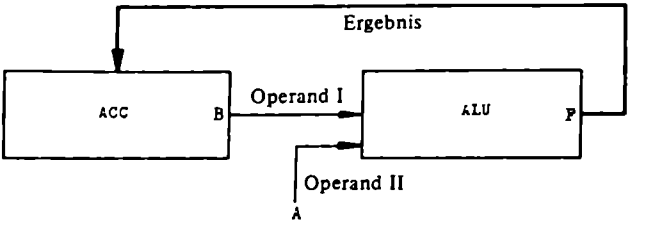


Bild 2.9: Akkumulator und ALU (Prinzip)

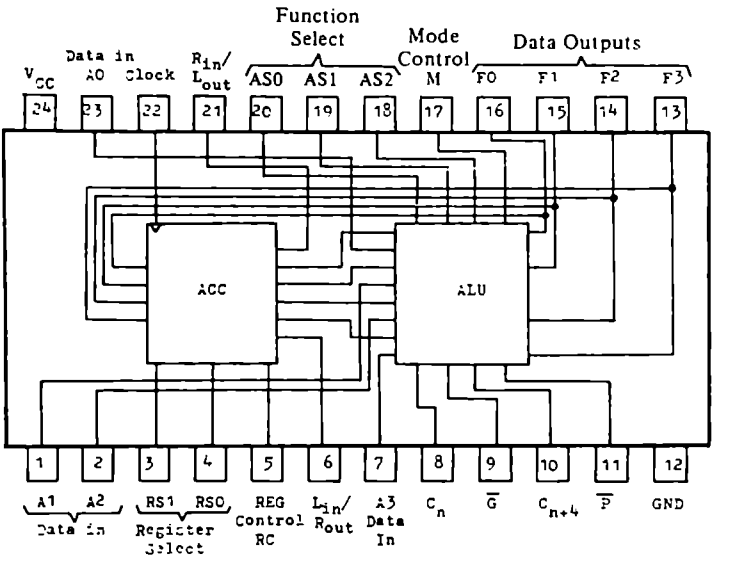


Bild 2.10: 4-Bit-Akkumulator und arithmetisch-logische Einheit (z.B. 74S281)

In Bild 2.10 ist der Schaltkreis einer 4-Bit-Akkumulator-Arithmetikeinheit dargestellt (74S281), die nach dem Prinzip von Bild 2.9 aufgebaut ist (Operand I = B; Operand II = A; Ergebnis = F_n). Die zu verarbeitenden Operanden können maximal 4 Bit breit sein. Am Befehlsvorrat (instruction set) wird gezeigt, daß allen Anforderungen, die in Abschnitt 2.3.1 an einen Befehlsvorrat einer ALU gestellt wurden, entsprochen werden kann.

Eine Erläuterung der einzelnen Ein- und Ausgänge der 4-Bit-Akkumulator-Arithmetikeinheit wird bei der Besprechung des Befehlsvorrats gegeben.

2.3.5.1 Befehlsvorrat der 4-Bit-Akkumulator-Arithmetikeinheit

Der Befehlsvorrat ist in Tabellenform in den Bildern 2.11 bis 2.12 aufgelistet. Drei verschiedene Befehlsarten sind möglich:

Arithmetische Befehle,
Logische Befehle,
Verschiebebefehle.

Jeder dieser Befehlsarten ist eine Tabelle zugeordnet.

Beziehungen zwischen dem jeweiligen Befehl und seinem Bitmuster, das an den Steuereingängen des Akkumulators und der ALU anliegt, kommt in den Tabellen zum Ausdruck. Die Besprechung der Befehle wird dies verdeutlichen.

Die arithmetischen Befehle (Bild 2.11)

Bei diesen Befehlen fällt auf, daß die Steuerleitung M (mode control = M) logisch „0" ist. Über die Steuerleitungen ASO, AS1 und AS2 werden in der ALU die ver-

schiedenen arithmetischen Befehle ausgewählt. Diese sind unter der Rubrik „Arithmetische Befehle F_n" in zwei getrennten Spalten, abhängig vom Incrementeingang (INC) angegeben. Mit F_n sind die Ausgänge der ALU bezeichnet ($F_n = F_3\ F_2\ F_1\ F_0$).

In diesem Zusammenhang sei darauf hingewiesen, daß sich die 4-Bit-Akkumulator-Arithmetikeinheit durch Parallelschalten von 4-Bit-Einheiten auf ein Vielfaches der Bitbreite der 4-Bit-Grundeinheit erweitern läßt. Deshalb ist der Name INC nur für die 4-Bit-Grundeinheit einer m · 4-Bit-Akkumulator-Arithmetikeinheit gültig, die die niedrigstwertigen Stellen, Bit 0 bis Bit 3, umfaßt.

Ansonsten muß diesem Eingang die Bedeutung eines Übertrag-(Carry-)Eingangs gegeben werden (C_n). Der Ausgang C_{n+4} gibt jeweils den Übertrag für die nächst höhere 4-Bit-Grundeinheit an. Der Index n ist definiert zu: n = 0, 4, 8 ... (4 · m). Die arithmetischen Befehle umfassen Addition ($F_n = A + B$), Subtraktion ($A - B, B - A$), Vermehren um Eins ($A + 1, B + 1$), Vermindern um Eins ($A - 1, B - 1$), Komplementieren ($\overline{B} = -B - 1$) und Negieren ($-A, -B$). Außerdem sind Befehle möglich, die einen Operanden unverändert lassen ($F_n = A$, $F_n = B$) oder die alle Ausgänge der ALU auf logisch „1" setzen.

Aus der Tabelle ist weiter zu entnehmen, daß es bis auf zwei Ausnahmen Befehle gibt, die sich entweder auf beide Operanden A und B oder nur auf einen der beiden Operanden A oder B anwenden lassen. Unter Operand A ist das 4-Bit-Binärmuster zu verstehen, welches über die Eingänge A_3 bis A_0 an der ALU anliegt. Operand B dagegen kommt aus dem Akkumulator. Er wird deshalb treffender als der Akkumulatorinhalt bezeichnet.

Ein Beispiel soll zeigen, wie ein bestimmter Befehl eingestellt wird.

Ein arithmetischer Befehl wird durch die Steuerleitungen AS0, AS1, AS2 und M sowie INC ausgewählt. Soll die Addition zweier Operanden A und B ausgeführt werden, so muß als Bitmuster AS0 = 1, AS1 = 1, AS2 = 0, M = 0 und INC = 0 eingestellt werden. Die ALU führt somit den Befehl $F_n = (A + B)$ aus, d.h. sie addiert die Eingangsinformation (A) und den Akkumulatorinhalt (B). Am Ausgang der ALU steht als F_n die Summe A + B. Die Frage, wie der Operand B in den Akkumulator kommt und das Ergebnis abgespeichert wird, soll bis zur Besprechung der Verschiebebefehle zurückgestellt werden.

Die logischen Befehle (Bild 2.11)

Für alle diese Befehle befindet sich die Steuerleitung M auf logisch „1". Die logischen Befehle umfassen folgende boolsche Verknüpfungen zweier Operanden A und B:

UND ($F_n = A \wedge B$), Inklusiv-ODER ($F_n = A \vee B$), Exklusiv-ODER ($F_n = A \veebar B$) sowie deren Negation ($F_n = \overline{A \wedge B}$; $F_n = \overline{A \vee B}$; $F_n = \overline{A \veebar B}$).

Bei diesen Befehlen spielt der Eingang INC keine Rolle, d.h. der anliegende Pegel ist uninteressant. In der Tabelle ist daher INC = X gesetzt worden. Darüber hinaus kann der Befehl $F_n = 0$ („Löschen") ausgeführt werden. Alle Ausgänge der ALU werden dadurch auf Null gesetzt.

Beispiel eines logischen Befehls:
Für den Befehl „UND (∧)" wird aus Bild 2.11 folgendes Bitmuster entnommen:

AS0 = 0, AS1 = 0, AS2 = 1, M = 1.

ALU-Steuerleitungen				Arithmetische Befehle F_n	
AS0	AS1	AS2	M	INC $C_0 = 0$	INC $C_0 = 1$
0	0	0	0	-2	-1
1	0	0	0	B - A - 1	B - A
0	1	0	0	A - B - 1	A - B
1	1	0	0	A + B	A + B + 1
0	0	1	0	B	B + 1
1	0	1	0	-B - 1	-B
0	1	1	0	A	A + 1
1	1	1	0	-A - 1	-A
ALU-Steuerleitungen				Logische Befehle F_n	
AS0	AS1	AS2	M	INC $C_0 = X$	
0	0	0	1	0	
1	X	0	1	$A \veebar B$	
0	1	0	1	$\overline{A \veebar B}$	
0	0	1	1	$A \wedge B$	
1	0	1	1	$\overline{A \vee B}$	
0	1	1	1	$\overline{A \wedge B}$	
1	1	1	1	$A \vee B$	

Bild 2.11: Befehlsvorrat der 4-Bit-Akkumulator-Arithmetikeinheit

Der Eingang INC hat auf den Befehl keinen Einfluß. Dieses Bitmuster legt in der ALU die boolsche Verknüpfung $A \wedge B$ fest, deren Ergebnis am Ausgang der ALU als $F_n = A \wedge B$ verfügbar ist. In derselben Weise lassen sich aus der Tabelle die Bitmuster für die anderen logischen Befehle entnehmen.

Die letzte Befehlsart, die Verschiebebefehle, gestattet die Beantwortung der Frage, wie Operanden und Ergebnisse in den Akkumulator kommen.

Die Verschiebebefehle (Bild 2.12)

Hatten es die arithmetischen und logischen Befehle mit der ALU zu tun, so werden die Verschiebebefehle im Akkumulator ausgeführt. Alle Verschiebebefehle bedingen jedoch ein besonderes Bitmuster an den Steuerleitungen der ALU und am INC-Eingang (INC = M = AS0 = AS1 = 0; AS2 = 1). Dieses Bitmuster wählt in der ALU den Befehl $F_n = B$ aus. Demnach wird der Akkumulatorinhalt B unverändert zum Aus-

gang der ALU durchgeschaltet (s. Tabelle Bild 2.12). Dieser Befehl $F_n = B$ hat die Eigenschaft eines NOP-Befehls (NOP = no operation, keine Veränderung des Operanden).

Die Wahl einzelner Verschiebebefehle erfolgt über bestimmte Bitmuster an den Steuereingängen RS0, RS1 und R_C des Akkumulators. Die in der Tabelle unter der Rubrik „Akkumulatorausgänge" dargestellte Information zeigt den Akkumulatorinhalt nach der Befehlsausführung an.

Parameter: INC = M = AS0 = AS1 = 0 AS2 = 1 $\Rightarrow Q_n = F_n = B_n$

Register Selektion RS0 RS1	Register-Steuereingang R_C	Clock-Eingang (aktive Flanke)	I_{in}/R_{out}	Akkumulatorausgänge MSB Q_D Q_C Q_B LSB Q_A	R_{in}/L_{out}	Verschiebebefehle
0 0	x	↑	z	f_3 f_2 f_1 f_0	z	$F_n \longrightarrow$ ACC (ACC laden)
1 0	0	↑	I_{in}	I_{in} Q_{Dn} Q_{Cn} Q_{Bn}	Q_{Bn}	Rechts schieben (logisch)
1 0	1	↑	I_{in}	Q_{DO} I_{in} Q_{Cn} Q_{Bn}	Q_{Bn}	Arithmetisch rechts schieben (Division mit Vorzeichen $-, + I_{in} = 1,0$)
0 1	0	↑	Q_{Cn}	Q_{Cn} Q_{Bn} Q_{An} r_{in}	r_{in}	Links schieben (logisch)
0 1	1	↑	Q_{Cn}	Q_{DO} Q_{Bn} Q_{An} r_{in}	r_{in}	Arithmetisch links schieben (Multiplikation mit Vorzeichen $r_{in} = 0$)
1 1	x	↑	z	Q_{DO} Q_{CO} Q_{BO} Q_{AO}	z	ACC bleibt unverändert
x x	x	0	x	Q_{DO} Q_{CO} Q_{BO} Q_{AO}	x	

Erläuterungen:

Rechts Schieben: $Q_D \longrightarrow Q_A$ Links Schieben: $Q_A \longrightarrow Q_D$

Z = Hochohmiger Zustand X = Eingang ohne Einfluß

$f_0, f_1, f_2, f_3, r_{in}, I_{in}$ = logischer Pegel an $F_0, F_1, F_2, F_3, R_{in}/L_{out}, R_{out}/L_{in}$

$Q_{AO}, Q_{BO}, Q_{CO}, Q_{DO}$ = logischer Pegel bevor R_{SO}, R_{S1}, R_C, Clock angelegt werden

$Q_{An}, Q_{Bn}, Q_{Cn}, Q_{Dn}$ = logischer Pegel bevor aktive Flanke des Clocks anliegt

X Eingang ohne Einfluß

Bild 2.12: Verschiebebefehle

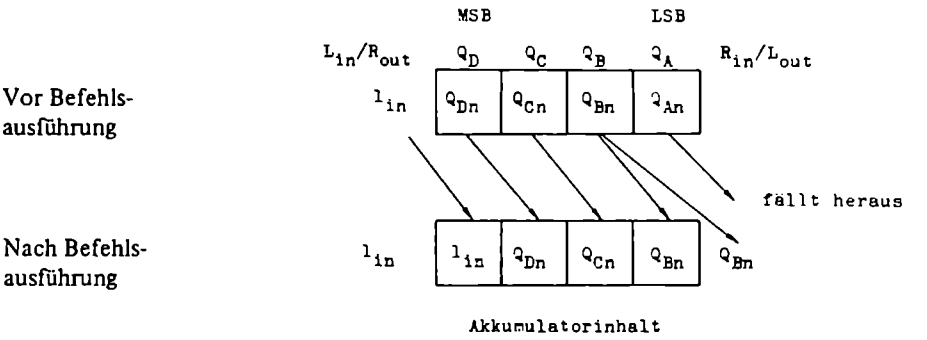
Die Verschiebebefehle lassen sich in drei verschiedene Gruppen einteilen:

Logische,
Arithmetische,
Sonstige.

Logische Verschiebebefehle

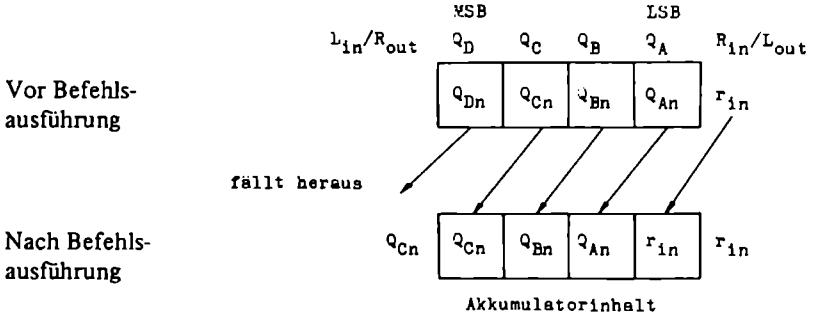
Diese Verschiebebefehle bewirken, daß der Akkumulatorinhalt um eine Stelle nach rechts bzw. nach links verschoben wird. Beide Verschiebebefehle lassen sich am besten an Bildern verdeutlichen. Die jeweiligen Bitmuster an den Steuereingängen RS0, RS1 und R_C sind in der Tabelle angegeben.

Rechts schieben



Der Akkumulatorinhalt wird um eine Stelle nach rechts verschoben, wobei der an L_{in}/R_{out} angelegte Wert ins MSB hereingeschoben wird und Q_{An} rechts herausgeschoben wird und verloren geht.

Links schieben



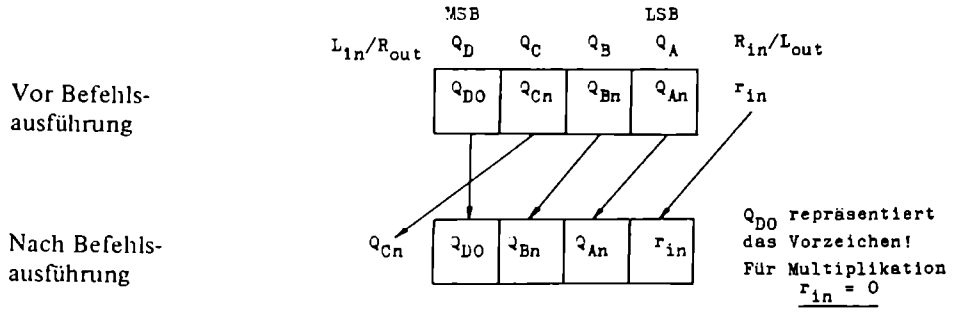
Der Akkumulatorinhalt wird um eine Stelle nach links verschoben, wobei der an R_{in}/L_{out} angelegte Wert ins LSB hineingeschoben wird und Q_{Dn} links herausgeschoben wird und verlorengeht. Diese Befehle eignen sich auch zur Multiplikation mit 2 (Links schieben, $r_{in} = 0$) und zur Division durch 2 (Rechts schieben, $l_{in} = 0$) von Dualzahlen ohne Vorzeichen (unsigned binary numbers).

Arithmetische Verschiebebefehle

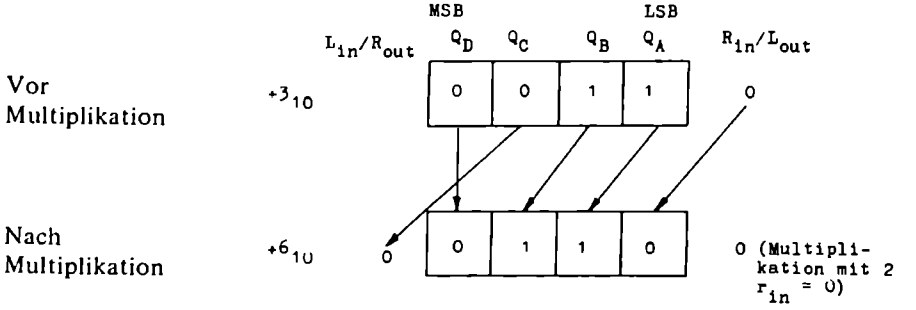
Wie schon der Name sagt, sind mit diesen Befehlen arithmetische Operationen verknüpft, die im Akkumulator durchgeführt werden. Diese Befehle bewirken, daß der Akkumulatorinhalt einer Multiplikation mit 2 oder einer Division durch 2 unterzogen werden kann. Bei den zu multiplizierenden und zu dividierenden Dualzahlen muß es sich um vorzeichenbehaftete handeln (signed binary numbers). Für die Beispiele wird ein Zahlenbereich wie folgt festgelegt:

Zahlenbereich $-8 \leqslant Z \leqslant 7$ wird vereinbart.

Arithmetisch links schieben (Multiplikation mit 2)

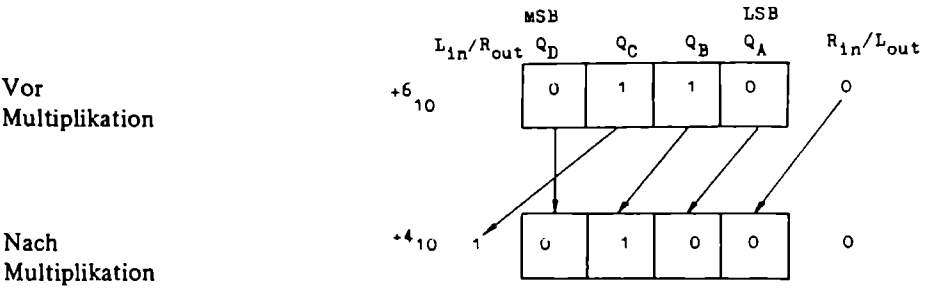


Beispiel: Multiplikation der positiven Dualzahl 0011_2 $(+3_{10})$ mit 2 (MSB = 0)

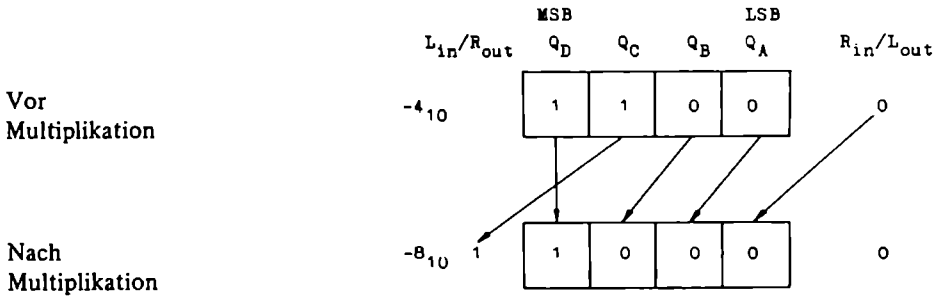


Die Multiplikation von $+3_{10}$ mit 2 ergibt vorzeichenrichtig $+6_{10}$. Wird jedoch $+6_{10}$ nochmals einer Multiplikation mit 2 unterzogen, so stellt sich ein falsches Ergebnis, nämlich $+4_{10}$ anstelle von $+12_{10}$ ein. Am vorliegenden Beispiel wird dies gezeigt.

Beispiel: Multiplikation der positiven Dualzahl 0110_2 $(+6_{10})$ mit 2

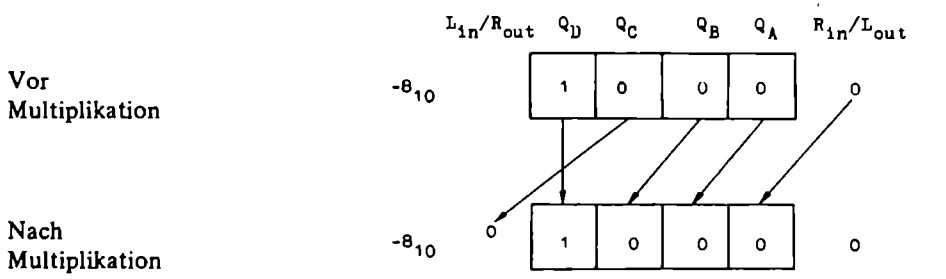


Beispiel: Multiplikation der negativen Dualzahl 1100_2 (-4_{10}) mit 2 (MSB = 1)



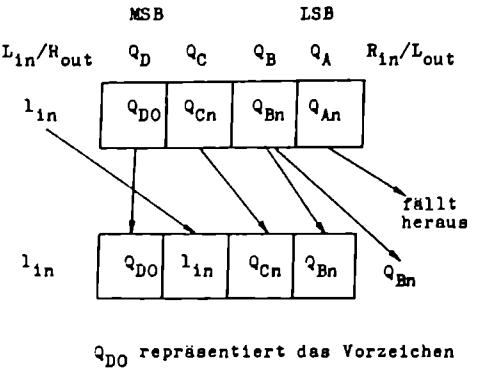
Die Multiplikation von -4_{10} mit 2 ergibt vorzeichenrichtig -8_{10}. Eine abermalige Multiplikation von -8_{10} mit 2 ergibt anstelle von -16_{10} das falsche Ergebnis -8_{10}.

Beispiel: Multiplikation der negativen Dualzahl 1000_2 (-8_{10}) mit 2

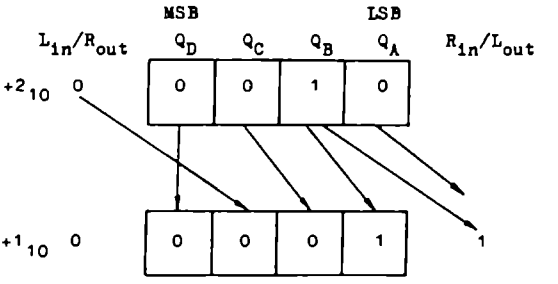


Die Überschreitung des Zahlenbereiches wird im Falle der Multiplikation vorzeichenbehafteter Dualzahlen durch Q_{Cn} angezeigt. Sobald Q_{Cn} = 1 (0) ist, liegt eine Zahlenbereichsüberschreitung bei positiven (negativen) Dualzahlen vor, sofern eine abermalige Multiplikation mit 2 ausgeführt wird.

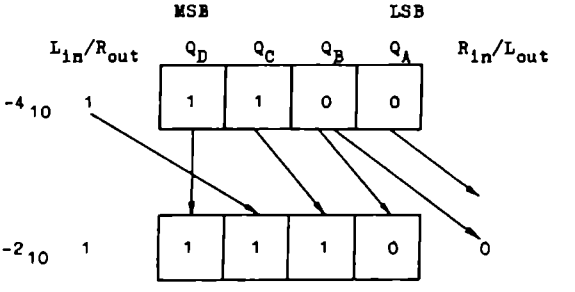
Arithmetisch rechts schieben (Division durch 2)



Beispiel: Division der positiven Dualzahl 0010_2 ($+2_{10}$) durch 2 (MSB = 0!)



Beispiel: Division der negativen Dualzahl 1100_2 (-4_{10}) durch 2 (MSB = 1!)



Bei der Division vorzeichenbehafteter Dualzahlen durch 2 muß l_{in} wie folgt gewählt werden:

positive Dualzahl $l_{in} = 0$
negative Dualzahl $l_{in} = 1$.

l_{in} ist demnach immer identisch mit dem Vorzeichenbit (sign bit = MSB).

Sonstige Verschiebebefehle

Unter diesen Befehlen seien diejenigen verstanden, die den Akkumulator laden oder den Akkumulatorinhalt nicht verändern. Der Befehl „ACC laden" leistet, daß die Information, welche am Ausgang der ALU ansteht, in den Akkumulator geladen, d.h. abgespeichert wird. Durch diesen Befehl wird dem Akkumulator seine wichtigste Aufgabe innerhalb der Akkumulator-Arithmetikeinheit zugeordnet, nämlich Zwischenspeicherung von Operanden. Der abgespeicherte Operand steht für weitere Befehle zur Verfügung. Die beiden noch verbleibenden Befehle lassen die Information des Akkumulators unverändert.

Zusammenfassend kann gesagt werden, daß mit der vorgestellten 4-Bit-Akkumulator-Arithmetikeinheit und ihrem Befehlsvorrat verdeutlicht wurde, welche Voraussetzungen gegeben sein müssen, um auf Operanden arithmetische, logische und Verschiebebefehle anwenden zu können. Weiter wurde abgeleitet, welche zentrale Bedeutung dem Akkumulator innerhalb der 4-Bit-Akkumulator-Arithmetikeinheit zukommt.

2.3.5.2 Erweiterung auf eine n · 4-Bit-Akkumulator-Arithmetikeinheit

Bei den behandelten arithmetischen und logischen Verschiebebefehlen sei darauf hingewiesen, daß sich im Falle einer Erweiterung auf eine n · 4-Bit-Akkumulator-Arithmetikeinheit (Bild 2.13) folgende Besonderheiten ergeben:

Werden logische Verschiebebefehle durchgeführt, also Rechts bzw. Links verschieben, so müssen an allen 4-Bit-Einheiten dieselben Bitmuster bezüglich RS0, RS1 und R_C angelegt werden.

Soll an der gleichen Einheit eine Multiplikation bzw. eine Division durchgeführt werden, so gilt:

Multiplikation

Arithmetisch links schieben nur auf die höchstwertige und niedrigstwertige 4-Bit-Einheit anwendbar (ansonsten logische Verschiebebefehle). In der höchstwertigen Einheit wird das Vorzeichenbit festgehalten und eine Zahlenbereichsüberschreitung angezeigt. R_{in} muß nur in der niedrigstwertigen Einheit zu logisch „0" gewählt werden. Bei allen anderen Übergängen zwischen zwei 4-Bit-Einheiten ist gemäß des logischen Verschiebebefehls „Links schieben" r_{in} festgelegt, weil bei zwei aufeinanderfolgenden 4-Bit-Einheiten R_{in}/L_{out} mit L_{in}/R_{out} verbunden ist.

Division

Arithmetisch rechts schieben nur auf höchstwertige 4-Bit-Einheit anwenden, ansonsten logische Verschiebebefehle „Rechts schieben" verwenden.

Dieser Sachverhalt wird durch Bild 2.13 verdeutlicht. Es zeigt auch die Zusammenschaltung mehrerer 4-Bit-Akkumulator-Arithmetikeinheiten und eines Look-Ahead Carry-Schaltkreises (74S182). Es wird bei dieser Konfiguration der Übertrag über jeweils 4 Bit nach dem Look-Ahead Carry-Verfahren gebildet. Dazu besitzt jede Einheit die Ausgänge $\overline{P}$ und $\overline{G}$.

Diese 4-Bit-Einheit wird in Bild 2.14 zu einem Rechenelement, bestehend aus Akkumulator (ACC) und arithmetisch-logischer Einheit (ALU) sowie einer Tastatur und einem Zwischenspeicher (7474) für die Ausgabe, erweitert. Über die Tastatur erfolgt die Eingabe der zu verarbeitenden Operanden, die Einstellung der ALU-Befehle über

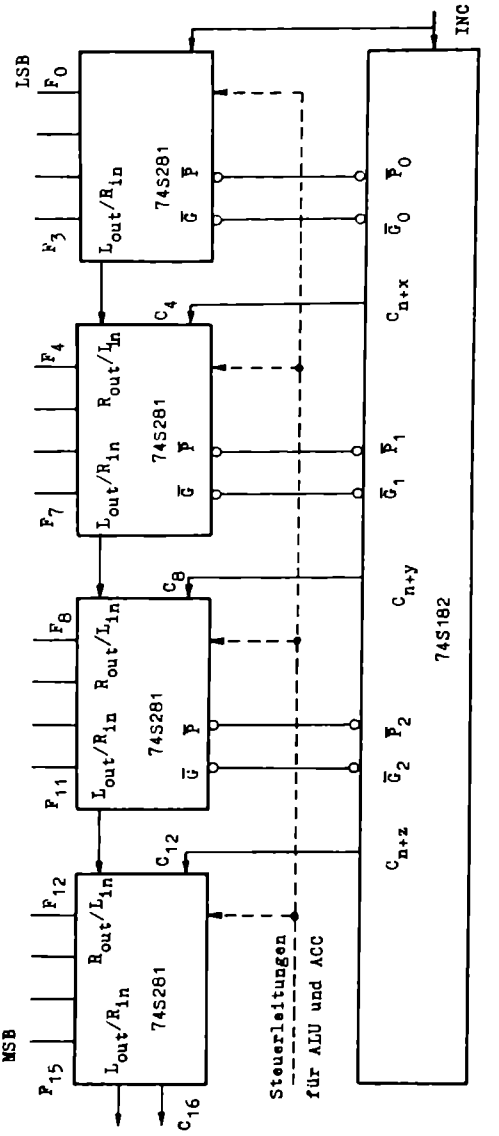


Bild 2.13: Erweiterte Arithmetikeinheit (16-Bit-Akkumulator-Arithmetikeinheit)

die Steuereingänge Mode Control, AS0, AS1, AS2 und INC sowie der Verschiebebefehle über RS0, RS1, R_C, L_{in}/R_{out}, R_{in}/L_{out} und die Ausgabe an die Anzeigeeinheit (Taste „Ausgeben").

Die Tastatur läßt sich in einen Eingabe- und einen Ausgabeteil sowie einen Steuerungsteil gliedern. Über letzteren wird die Funktion des Rechenelements festgelegt. Im weiteren Verlauf dieses Kapitels wird gezeigt, wie aus einem Bitmuster, Operationscode

(**operationcode** = Op-Code) genannt, die Information für diese Steuereingänge abgeleitet werden kann, um einen Befehl mit dem Rechenelement auszuführen.

Zunächst wird die Funktion des Rechenelements anhand eines Beispiels erläutert.

Es soll die folgende Operation durchgeführt werden:
Man addiere die positiven 4-Bit-Dualzahlen X und Y, erhöhe das Ergebnis um Eins und speichere es im Akkumulator ab. Man dividiere dieses Ergebnis durch Zwei und gebe es an die Anzeigeeinheit aus.

Gegeben seien $X = 0011_2$ und $Y = 0010_2$.

Die Behandlung des Beispiels (X + Y + 1) : 2 erfordert eine Zerlegung in einzelne Befehlsschritte.

1. Schritt
Eingabe von $X = 0011_2$ in den Akkumulator.
Einstellen des Bitmusters (vgl. Bild 2.11):

Eingabe				Steuereingänge für ALU und ACC							
A_3	A_2	A_1	A_0	AS0	AS1	AS2	M	INC	RS0	RS1	R_C
0	0	1	1	0	1	1	0	0	0	0	X

Mit Betätigung der Clock-Taste steht X im Akkumulator.

2. Schritt
Addition von $Y = 0010_2$ zum Akkumulatorinhalt X.
Einstellen des Bitmusters (vgl. Bild 2.11):

Eingabe				Steuereingänge für ALU und ACC							
A_3	A_2	A_1	A_0	AS0	AS1	AS2	M	INC	RS0	RS1	R_C
0	0	1	0	1	1	0	0	0	0	0	X

Mit Betätigung der Clock-Taste wird die Summe X + Y im Akkumulator abgespeichert.

3. Schritt
Erhöhen des Akkumulatorinhalts um Eins.
Einstellen des Bitmusters (vgl. Bild 2.11):

Eingabe				Steuereingänge für ALU und ACC							
A_3	A_2	A_1	A_0	AS0	AS1	AS2	M	INC	RS0	RS1	R_C
X	X	X	X	0	0	1	0	1	0	0	X

Nach Betätigung der Clock-Taste steht im Akkumulator (X + Y + 1).

4. Schritt

Division des Akkumulatorinhalts durch Zwei und Ausgabe des Ergebnisses.
Einstellen des Bitmusters (vgl. Bild 2.12):

Eingabe				Steuereingänge für ALU und ACC								
A_3	A_2	A_1	A_0	AS0	AS1	AS2	M	INC	RS0	RS1	R_C	l_{in}
X	X	X	X	0	0	1	0	0	1	0	1	0

Am Ausgang des Zwischenspeichers steht das Ergebnis (X + Y + 1) : 2 an, sobald die Clock-Taste und die Taste „Ausgeben" betätigt werden.

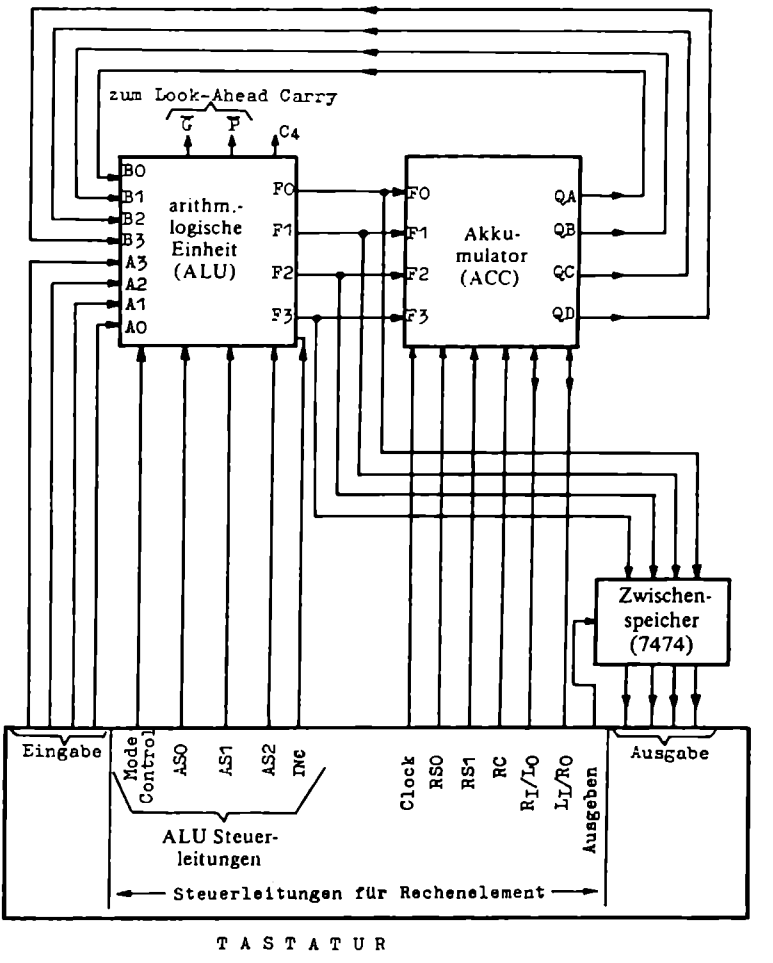


Bild 2.14: 4-Bit-Rechenelement (ALU und ACC)

Die Ergebnisse und die angelegten Bitmuster sind für das Beispiel in der nachfolgenden Tabelle in Bild 2.15 zusammengestellt.

An dieser Tabelle wird nochmals deutlich, daß die Rechenoperation (X + Y + 1) : 2 in einzelne Schritte zerlegt werden muß, die nacheinander auszuführen sind. Jeder Schritt seinerseits zerfällt wieder in weitere Teilschritte, da z.B. ein Ergebnis, bevor es in den Akkumulator abgespeichert wird, in der ALU berechnet werden muß.

Eingabe	ALU-Steuer-leitungen	ACC-Steuer-leitungen	Clock-Taste	Taste	Akkumulator-inhalt
A_3 A_2 A_1 A_0	AS0 AS1 AS2 M INC	RS0 RS1 R_C 1_{in}		Ausgeben	Q_D Q_C Q_B Q_A
0 0 1 1	0 1 1 0 0	0 0 X X	↓	0	0 0 1 1
0 0 1 0	1 1 0 0 0	0 0 X X	↓	0	0 1 0 1
X X X X	0 0 1 0 1	0 0 X X	↓	0	0 1 1 0
X X X X	0 0 1 0 0	1 0 1 0	↓	↓	0 0 1 1

Bildung von (X+Y+1):2 mit Y=0010$_2$ und X=0011$_2$

X Eingang ohne Einfluß

↓ aktive Flanke (Übergang von logisch "0" nach "1")

Bild 2.15: Funktionsbeispiel zum Rechenelement

2.3.6 Die Register-Arithmetikeinheit

Bei der besprochenen Akkumulator-Arithmetikeinheit konnte nur ein Operand im Akkumulator zwischengespeichert werden. Des öfteren besteht jedoch die Aufgabe, mehr als einen Operanden abzulegen, d.h. von verschiedenen Befehlen müssen Zwischenergebnisse für die spätere Weiterverarbeitung abgespeichert werden können.

Dieser Anforderung entspricht eine in Bild 2.16 dargestellte Register-Arithmetikeinheit, deren Hauptunterschied zur Akkumulator-Arithmetikeinheit darin besteht, daß sie 16 Akkumulatoren besitzt. Diese Erweiterung wird mittels eines 64-Bit-(16 x 4-) Schreib-Lese-Speichers erreicht (74S189). Ansonsten besteht die Register-Arithmetikeinheit noch aus einer ALU (74S181), einem Multiplexer mit Zwischenspeicher (74298) sowie einem Schieberegister und einem Zwischenspeicher (7474), der die Ausgabe der Ergebnisse an die Peripherie ermöglicht. Die Anordnung und Auswahl der Bausteine wurde bewußt gegenüber Bild 2.14 abgeändert, um eine weitere Möglichkeit zum Aufbau einer Akkumulator-Arithmetikeinheit aufzuzeigen. Die Erweiterung auf 16 Akkumulatoren macht eine Ergänzung der Definition des Begriffes Akkumulator aus Abschnitt 2.3.5 notwendig. Der Akkumulator war als Register de-

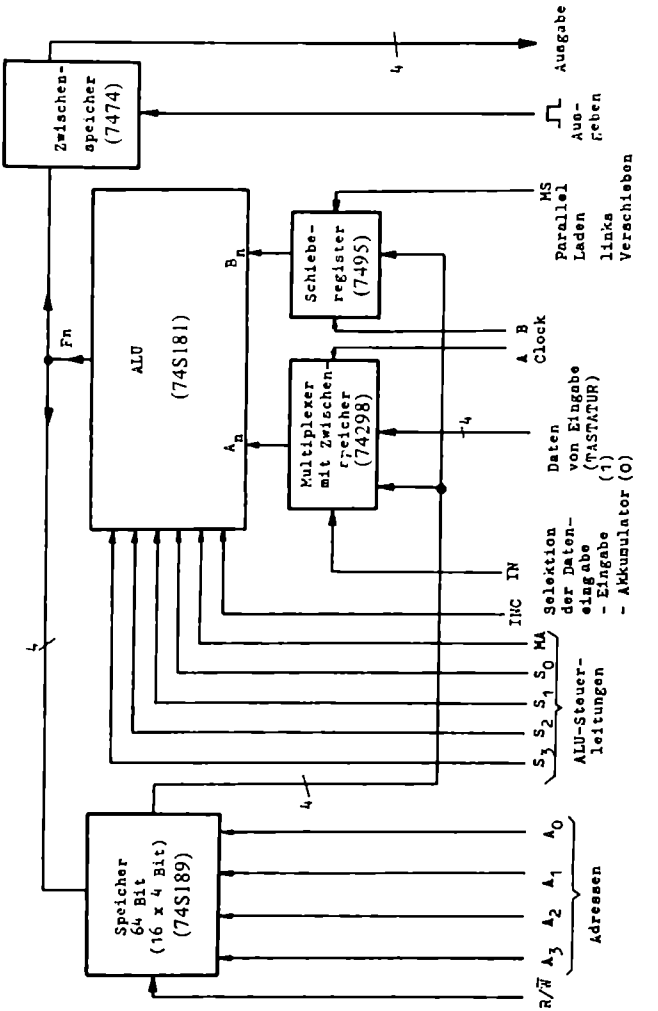


Bild 2.16: 4-Bit-Register-Arithmetikeinheit

finiert, das der Speicherung von Operanden diente und Verschiebebefehle ermöglichte. Aus der Forderung nach mehr Möglichkeiten, um Operanden zwischenzuspeichern, entsprang der Gedanke, die Anzahl der Akkumulatoren zu erhöhen. Da nur zusätzlicher Speicherplatz benötigt wird, können dazu Register verwendet werden. Dadurch erscheint eine Aufteilung in Speicherplätze für Operanden und ein Schieberegister (7495) für die Verschiebebefehle sinnvoll. Jedes der 16 Register hat Zugriff zum Schieberegister und kann somit im Sinne der Definition 1 aus Abschnitt 2.3.5 als Akkumulator bezeichnet werden.

Durch Ansteuerung der Adreßleitungen A_0 bis A_3 des Schreib-Lese-Speichers kann einer der 16 Akkumulatoren ausgewählt werden, so daß über das Schieberegister und den Multiplexer Operanden von den Akkumulatoren an den Eingängen A_n und B_n der ALU bereitgestellt werden können. Über den Multiplexer können außerdem Operanden über eine Eingabe an die ALU angelegt werden. Diese hat die Möglichkeit, gesteuert von den Funktionsselektionseingängen, arithmetische und logische Befehle auszuführen. Das Abspeichern von Ergebnissen in einen Akkumulator wird über die R/$\overline{W}$-Leitung gesteuert (Read/$\overline{Write}$ = R/$\overline{W}$), die dafür in den logischen Zustand „Schreiben" ($\overline{Write}$ ≙ logisch „0") gebracht wird.

Die Register-Arithmetikeinheit wird durch einige zusätzliche Schaltkreise erweitert, so daß es möglich wird, über eine Tastatur Befehle einzustellen und Operanden ein- und auszugeben (Bild 2.17). In diesem Zusammenhang bedarf es noch der Erklärung der Begriffe Ursprungsort (source), Bestimmungsort (destination) und Abspeichern (deposit). Unter dem Ursprungsort und dem Bestimmungsort sind gleicherweise Akkumulatoren zu verstehen. Beide Begriffe definieren einen bestimmten Platz innerhalb des Registerbereiches (= 16 Akkumulatoren). Dabei bezeichnet der *Ursprungsort, woher* ein Operand kommt, und der *Bestimmungsort, wohin* ein Ergebnis abgespeichert werden soll. Ursprungsort und Bestimmungsort werden über eine Adresse (A_3 bis A_0) festgelegt. Über den Eingang „Abspeichern" wird erreicht, daß ein Ergebnis, das an den Ausgängen der ALU ansteht, an einem ausgewählten Bestimmungsort abgespeichert, d.h. eingeschrieben wird. Dies bedingt, daß am Eingang „Abspeichern" eine logische „0" anliegt (R/$\overline{W}$-Leitung im Zustand $\overline{Write}$). Sind in einem Befehl Ursprungsort und Bestimmungsort identisch, liegt ein Einadreßbefehl vor. Ein Zweiadreßbefehl ist gegeben, sobald Ursprungsort und Bestimmungsort verschieden sind. Dieser Sachverhalt wird am Befehlsvorrat der Register-Artithmetikeinheit deutlich. Der Befehlsvorrat läßt sich aus dem der ALU (74S181) und dem des Schieberegisters (SR) ableiten. Für beide Schaltkreise sind die jeweiligen Befehle in den Bildern 2.18 und 2.19 zusammengestellt. Die im Befehlsvorrat der Register-Arithmetikeinheit realisierten Befehle sind so ausgewählt worden, daß sie den Anforderungen einer ALU entsprachen. Dies ergab einen Befehlsvorrat, der aus 16 verschiedenen Befehlen besteht. Jedem dieser Befehle entspricht – von den Steuerleitungen für ALU und SR her gesehen – ein bestimmtes, nur für diesen Befehl gültiges Bitmuster. Um die Funktion der ALU und des SR festzulegen, werden 8 Steuerleitungen benötigt. Die 16 Befehle bedingen 16 verschiedene Bitmuster. Diese werden an 16 Speicherplätzen, die aus je 8 Stellen (Bits) bestehen, in einem Festwertspeicher (read only memory = ROM) abgelegt. Den 16 Speicherplätzen sind 16 Adressen zugeordnet. Jeder der 16 Adressen ist über deren Inhalt ein Befehl zugeordnet. Die 16 Adressen können ihrerseits in binärer Form dargestellt werden. Dieser Code wird Befehlscode (operationcode = Op-Code) genannt. Dieser Befehlscode kann, da 16 Speicher-

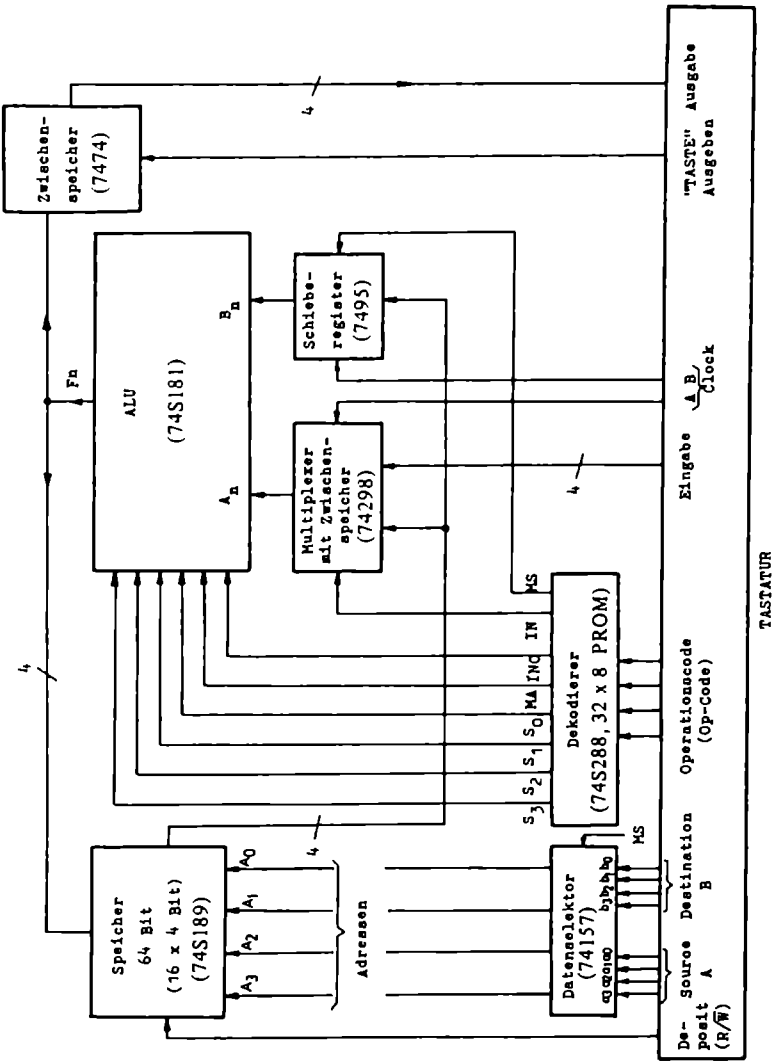


Bild 2.17: Erweiterte 4-Bit-Register-Arithmetikeinheit

plätze adressiert werden müssen, mit einer 4-Bit-Dualzahl verwirklicht werden. Der Befehlsvorrat ist in Tabellenform in Bild 2.20 dargestellt. Er umfaßt arithmetische, logische und Ein-/Ausgabebefehle. Die Befehle können je nach Befehlsart mit Angaben über den Ursprungsort oder Bestimmungsort der Operanden bzw. Ergebnisse versehen werden. Diese sind in den betreffenden Rubriken „Ursprungsort", „Bestimmungsort" enthalten. In den weiter rechts stehenden Rubriken sind der Op-Code, das Bitmuster für die ALU und das SR sowie eine Abkürzung für den jeweiligen Befehl angegeben. Die Abkürzung eines Befehls wird häufig als *Mnemonic* bezeichnet. Dieser Begriff wird im Kapitel 4 im Zusammenhang mit dem Befehlsvorrat eines

ALU-Steuerleitungen					Arithmetische Befehle	
S3	S2	S1	S0	MA	INC = 1	INC = 0
0	0	0	0	0	*A + 1	*A
0	0	0	1	0	(A∨B) + 1	A∨B
0	0	1	0	0	(A∨$\overline{B}$) + 1	A∨$\overline{B}$
0	0	1	1	0	*0	*−1
0	1	0	0	0	A + (A∧$\overline{B}$) + 1	A + (A∧$\overline{B}$)
0	1	0	1	0	(A∨B) + (A∧$\overline{B}$) + 1	(A∨B) + (A∧$\overline{B}$)
0	1	1	0	0	*A − B	A − B − 1
0	1	1	1	0	A∧$\overline{B}$	(A∧$\overline{B}$) − 1
1	0	0	0	0	A + (A∧B) + 1	A + (A∧B)
1	0	0	1	0	A + B + 1	*A + B
1	0	1	0	0	(A∨$\overline{B}$) + A∧B + 1	(A∨$\overline{B}$) + A∧B
1	0	1	1	0	A∧B	(A∧B) − 1
1	1	0	0	0	A + A + 1	*A + A
1	1	0	1	0	(A∨B) + A + 1	(A∨B) + A
1	1	1	0	0	(A∨$\overline{B}$) + A + 1	(A∨$\overline{B}$) + A
1	1	1	1	0	*A	*A − 1

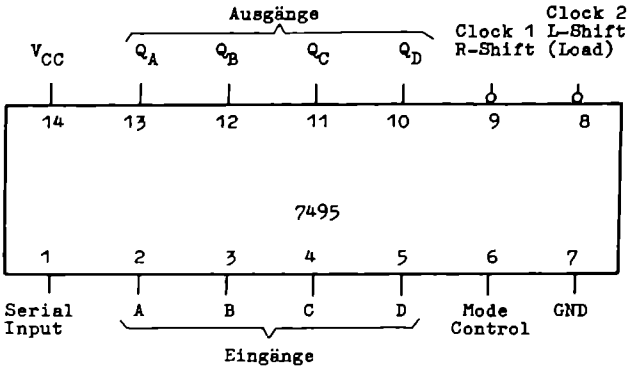
ALU-Steuerleitungen					Logische Befehle
S3	S2	S1	S0	MA	INC = X
0	0	0	0	1	*$\overline{A}$
0	0	0	1	1	$\overline{A∨B}$
0	0	1	0	1	$\overline{A}$∧B
0	0	1	1	1	0
0	1	0	0	1	$\overline{A∧B}$
0	1	0	1	1	$\overline{B}$
0	1	1	0	1	A⊻B
0	1	1	1	1	A∧$\overline{B}$
1	0	0	0	1	$\overline{A}$∨B
1	0	0	1	1	$\overline{A⊻B}$
1	0	1	0	1	*B
1	0	1	1	1	*A∧B
1	1	0	0	1	*1
1	1	0	1	1	A∨$\overline{B}$
1	1	1	0	1	*A∨B
1	1	1	1	1	A

* Diese Funktionen werden beim Befehlsvorrat der Register-Arithmetikeinheit verwendet
X Eingang ohne Einfluß

−A = $\overline{A}$ + 1

Bild 2.18: Befehlsvorrat der 4-Bit-ALU (74S181)

Mikroprozessors Verwendung finden und noch näher erklärt werden. Unter der Rubrik „Klammerschreibweise" findet sich eine Darstellung, die die Funktion eines Befehls eindeutig beschreibt. Der Befehl wird in seinem Ablauf, d.h. wie er ausgeführt wird und was dabei geschieht, kurz gekennzeichnet. Insbesondere wird deutlich, woher die Operanden kommen und wohin das Ergebnis gebracht wird. Die Operanden



Mode Control (MS)	Clock-Eingänge (C1, C2)	Eingänge D C B A	Ausgänge Q_D Q_C Q_B Q_A	Funktion
0	↓	d Q_D Q_C Q_B	d Q_{Dn} Q_{Cn} Q_{Bn}	Logisch rechts Schieben
1	↓	d c b a	d c b a	Parallel laden
1	1	X X X X	Q_{DO} Q_{CO} Q_{BO} Q_{AO}	Schieberegister hält Information fest

Indizes 0, n siehe Erläuterung Bild 2.12

X Eingang ohne Einfluß

↓ Aktive Flanke (Übergang logisch "1" nach "0")

Bild 2.19: Befehlsvorrat des Schieberegisters

kommen zum einen aus den Akkumulatoren und zum anderen können sie über die Tastatur von „außen" eingegeben werden (siehe Befehl ADD und Befehl INPUT). Ergebnisse werden entweder in einem der Akkumulatoren abgespeichert, oder sie werden über die Ausgabe an die Tastatur „ausgegeben". Die Aufgabe, die ALU und das SR mit den zu verarbeitenden Operanden zu versorgen, sie zu verarbeiten und die gewonnenen Ergebnisse abzuspeichern oder auszugeben, wird von den Steuereingängen Clock A, Clock B, R/$\overline{W}$ und „Ausgeben" übernommen. Die einzelnen Vorgänge, die dabei ablaufen und die Abarbeitung eines Befehls möglich machen, werden als Mikrozyklen oder Zustände (states) bezeichnet. Diese Begriffe werden bei der Behandlung des nachfolgenden Beispiels noch an Anschaulichkeit gewinnen.

An einem Beispiel soll die Funktionsweise der Register-Arithmetikeinheit veranschaulicht werden. Es sei wiederum als Beispiel eine Addition gewählt.

Annahme: Die beiden zu addierenden Operanden seien in den Akkumulatoren ACCA und ACCB abgelegt.

Ursprungsort (Source) $a_3\ a_2\ a_1\ a_0$	Bestimmungsort (Destination) $b_3\ b_2\ b_1\ b_0$	Operationscode (Op-Code) (oktal)	Steuerleitungen für ALU und SR S3	S2	S1	S0	MA	INC	IN	MS	Befehl	Klammerschreibweise	Clock A	Clock B	R/$\overline{W}$	Ausgeben
A	A	0	0	0	0	0	0	0	0	1	NOP	⟨ACCA⟩→ACCA	↓	1	↓	0
A	A	0	0	0	0	0	0	0	0	1	Multiplexer laden	⟨ACCA⟩→MUX	↓	1	1	0
A	B	0	0	0	0	0	0	0	0	1	MOV	⟨ACCA⟩→ACCB	↓	1	↓	0
A	OUTPUT	1	1	1	1	1	1	X	0	0	OUT	⟨ACCA⟩→OUTPUT	↓	1	1	↑
A	A	2	0	0	1	1	0	1	0	0	CLR	0 →ACCA	↓	1	↓	0
A	A	3	0	0	1	1	0	X	0	0	SET	−1 →ACCA	↓	1	↓	0
A	A	4	0	0	0	0	0	1	0	0	INC	⟨ACCA⟩+1 →ACCA	↓	1	↓	0
A	A	5	1	1	1	1	0	0	0	0	DEC	⟨ACCA⟩−1→ACCA	↓	1	↓	0
A	A	6	0	0	0	0	1	X	0	0	COM	$\overline{\langle ACCA\rangle}$→ACCA	↓	1	↓	0
A	B	7	1	0	0	1	0	0	0	1	ADD	⟨ACCA⟩+⟨ACCB⟩→ACCB	1	↓	↓	0
A	B	10	0	1	1	0	0	1	0	1	SUB	⟨ACCB⟩−⟨ACCA⟩→ACCB	1	↓	↓	0
A	B	11	1	1	1	0	1	X	0	1	IOR	⟨ACCA⟩∨⟨ACCB⟩→ACCB	1	↓	↓	0
A	B	12	0	1	1	0	1	X	0	1	EOR	⟨ACCA⟩∀⟨ACCB⟩→ACCB	1	↓	↓	0
INPUT	B	13	1	1	1	1	0	1	1	1	IN	⟨INPUT⟩→ACCB	1	↓	↓	0
A	B	14	1	0	1	1	1	X	0	1	AND	⟨ACCA⟩∧⟨ACCB⟩→ACCB	1	↓	↓	0
INPUT	B	15	1	0	1	1	1	X	1	1	MASK	⟨INPUT⟩∧⟨ACCB⟩→ACCB	1	↓	↓	0
A	A	16	1	1	0	0	0	0	0	1	ASL	⟨ACCA⟩x 2^1 →ACCA	↓	1	↓	0
A	A	17	1	0	1	0	1	X	0	0	ASR	⟨ACCA⟩x 2^{-1} →ACCA	1	↓	↓	0

Bemerkung: X Eingang ohne Einfluß
Bei Zweiadreßbefehlen erfolgt das Laden des Schieberegisters und die Ausführung der Operation nacheinander.
Bei dem Befehl IN kann eine UND-Verknüpfung zwischen Daten, die von der Peripherie oder vom Datenspeicher bzw. einem Akkumulator kommen, durchgeführt werden.
ACCA und ACCB können als Source bzw. Destination einer von 16 Akkumulatoren sein (z.B. ACC7, dann sind $A_3 = 0$, $A_2 = 1$, $A_1 = 1$, $A_0 = 1$).

Bild 2.20: Befehlsvorrat der 4-Bit-Register-Arithmetikeinheit

Befehlsablauf

1. Schritt (vgl. Bild 2.20)

Operand A wird in den Zwischenspeicher des Multiplexers geladen.

Dazu muß der Akkumulator A (ACCA) adressiert, der Ladebefehl ausgewählt und der Akkumulatorinhalt ((ACCA)) in den Zwischenspeicher des Multiplexers geladen werden. Letzteres übernimmt Clock A.

2. Schritt (vgl. Bild 2.20)

Operand B wird ins Schieberegister geladen und die Addition (ACCA) + (ACCB) wird ausgeführt.

Adressieren von Akkumulator B (ACCB) und Laden des Schieberegisters (SR) mit Inhalt von Akkumulator B ((ACCB)) mittels Clock B. Einstellen des Befehls ADD, indem der Op-Code 7_8 über die Tastatur an den Festwertspeicher (read only memory = ROM) angelegt wird. Die ALU kann jetzt die Addition von (ACCA) und (ACCB) durchführen.

3. Schritt (vgl. Bild 2.20)

Abspeichern des Ergebnisses nach ACCB.

Der Akkumulator B (ACCB) wird adressiert, und sobald die Steuerleitung $R/\overline{W}$ auf logisch „0" geht, wird die Summe ((ACCA) + (ACCB)) in ACCB abgespeichert.

Schlußfolgerungen:

Der ausgeführte Befehl hat gezeigt, daß der Bestimmungsort für das Ergebnis gleichzeitig der Ursprungsort eines der beiden Operanden ist. Beim Befehlsvorrat eines Mikroprozessors wird dieser Sachverhalt ebenfalls anzutreffen sein. Wie schon am Beispiel der Akkumulator-Arithmetikeinheit festgestellt wurde, so trifft auch hier die Bemerkung zu: Befehle, selbst einfache wie eine Addition, erfordern eine genaue Steuerung der einzelnen Befehlsschritte, um die zeitlich richtige Verarbeitung der Operanden sicherzustellen (Bild 2.21).

Es liegt nun der Schluß nahe, diese manuelle Steuerung der Register-Arithmetikeinheit einer speziell dafür konzipierten Baugruppe innerhalb der Zentraleinheit (CPU) als Aufgabe zu übertragen. Diese Baugruppe wird Steuerwerk (control unit = CU) genannt.

Zunächst jedoch wird das Prinzip einer busorientierten Zentraleinheit vorgestellt, die in ihrem Aufbau für viele Mikroprozessoren typisch ist. An diesem Beispiel wird gezeigt, wie das Steuerwerk innerhalb der Zentraleinheit arbeitet.

2.3.7 Die busorientierte Zentraleinheit

2.3.7.1 Grundlegendes zum Aufbau

In Bild 2.22 ist eine busorientierte Zentraleinheit wiedergegeben. Sie kann durch einfaches Umzeichnen aus der Register-Arithmetikeinheit hergeleitet werden. Neu hinzugekommen sind das Steuerwerk mit dem Befehlsregister sowie das Statusregister und ein externer Taktgenerator. Als Register der Zentraleinheit sind der Vollständigkeit halber der Programmzähler, das Indexregister, der Stack Pointer, eine Anzahl Akku-

Source A	Destination B	Operations code (Op-Code)	Clock A	Clock B	Deposit (R/$\overline{W}$)	Befehlsschritt
$a_3\ a_2\ a_1\ a_0$	X X X X	0 0 0 0	↓	1	1	Zwischenspeicher im Multiplexer mit <ACCA> laden
X X X X	$b_3\ b_2\ b_1\ b_0$	0 1 1 1	1	↓	1	Schieberegister laden und Addition von <ACCA> und <ACCB>
X X X X	$b_3\ b_2\ b_1\ b_0$	0 1 1 1	1	1	↓	Abspeichern des Ergebnisses nach ACCB

Befehl: <ACCA> + <ACCB> ⟶ ACCB

Bild 2.21: Funktionsbeispiel zur 4-Bit-Register-Arithmetikeinheit

mulatoren, das Datenregister und ein Adreßregister aufgeführt. Das Indexregister und der Stack Pointer werden im Kapitel 4 noch ausführlich erklärt.

Wichtig ist bei dieser Anordnung, daß alle Register mit der arithmetisch-logischen Einheit über gemeinsame Leitungen, einen sogenannten BUS, verbunden sind. Dies hat zur Folge, daß alle Register, um die Arithmetik einfach zu gestalten, eine Bitbreite haben, die derjenigen der arithmetisch-logischen Einheit entspricht. An dieser Stelle soll gesagt werden, daß durchaus auch verschiedene Bitbreiten zwischen Akkumulator und arithmetisch-logischer Einheit einerseits und dem Programmzähler, dem Indexregister und dem Stack Pointer andererseits üblich sind. An der Zusammenschaltung von Zentraleinheit und Speicher wird gezeigt werden, daß aufgrund der verschiedenen Funktionen, die die Register zu erfüllen haben, nämlich Adressierung eines Speicherplatzes und Datenaustausch zwischen Zentraleinheit und Speicher, obige Aussage bestätigt werden kann. Ein Datenaustausch ist auf Leitungen angewiesen, über die er erfolgen, überwacht und gesteuert werden kann. Dazu sind in der busorientierten Zentraleinheit die Daten- und Adreßleitungen sowie die Steuerleitungen vorgesehen. Ihre Aufgaben werden anhand eines Mikroprozessorsystems oder Mikrocomputers ausführlich erläutert. Ebenfalls werden die für einen Datenaustausch wichtigen Register der Zentraleinheit (Programmzähler, Datenregister, Adreßregister) erklärt.

In Bild 2.22 kommt einem anderen Teil der Zentraleinheit eine wichtige Bedeutung zu, nämlich dem Steuerwerk (CU). Die Aufgaben des Steuerwerks werden im folgenden betrachtet.

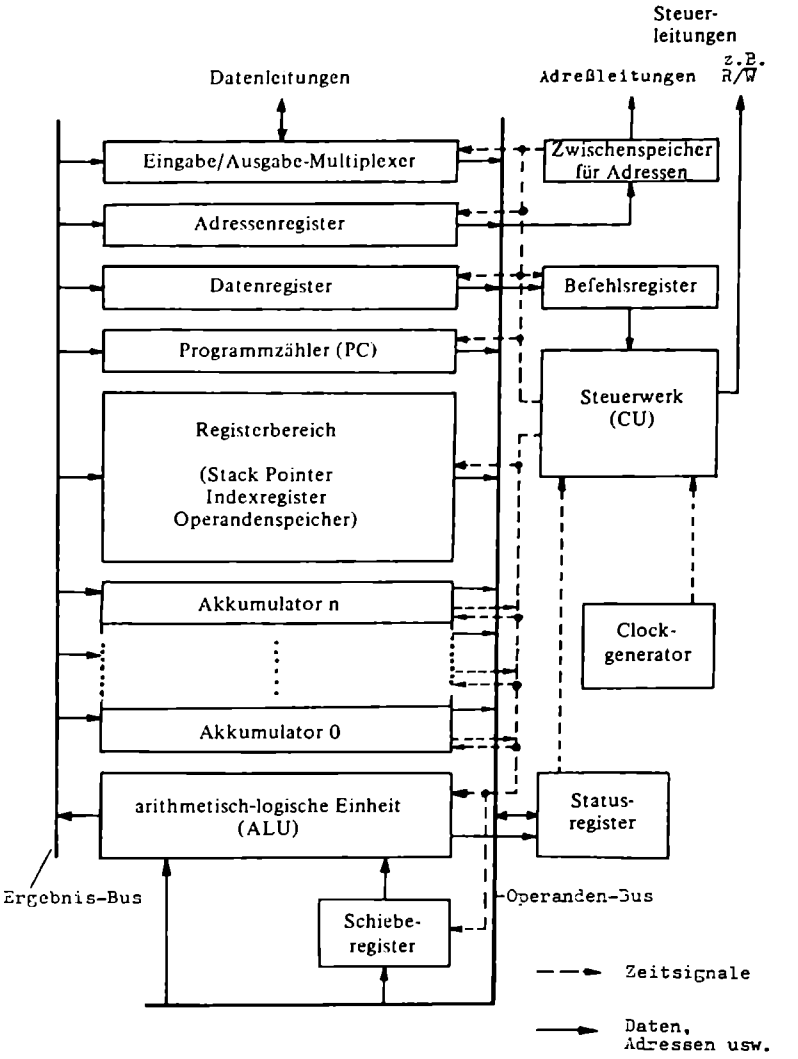


Bild 2.22: Busorientierte Zentraleinheit

2.3.7.2 Aufgaben des Steuerwerks

Die Notwendigkeit eines Steuerwerks, oft auch Leitwerk genannt, ergab sich aus den mit der Register-Arithmetikeinheit und der Akkumulator-Arithmetikeinheit durchgeführten Beispielen. Diese machten anhand der manuellen Ausführung der Befehle, deren Zerlegung in einzelne Befehlsschritte, deutlich, daß es eine Aufgabe der zu konzipierenden Baugruppe sein müßte, gerade diese Vorgänge zu steuern. Damit ist bereits eine wesentliche Aufgabe des Steuerwerks erkannt, nämlich für eine

zeitlich richtige Ausführung eines Befehls zu sorgen. Darunter ist die Bereitstellung der Operanden, ihre Verknüpfung in der ALU und schließlich das Abspeichern des Ergebnisses zu verstehen.

Der eigentliche Vorzug des Steuerwerks zeigt sich, sobald folgendes für einen Mikroprozessor typische Problem zu lösen ist:

Eine Folge von verschiedenen Befehlen soll nacheinander ausgeführt werden. Dies heißt mit anderen Worten, die Zentraleinheit hat die Aufgabe, ein Programm abzuarbeiten.

Das Steuerwerk leistet in diesem Fall:

- Steuerung der Ein- und Ausgabe von Operanden, Ergebnissen und Befehlen, des Transfers von und zur Peripherie, des Transfers von und zum Speicher.
- Steuerung aller arithmetischen und logischen Befehle der ALU (Adressenberechnung und Arithmetik).
- Dekodierung eines Befehls, d.h. Zerlegung desselben in eine Folge von seriellen und parallelen Anweisungen, meistens unter Zuhilfenahme eines externen Taktes. Diese Anweisungen haben Operationen in den einzelnen Registern und der ALU zur Folge (→ Ausführung eines Befehls → Mikrooperationen siehe Kapitel 7: Mikroprogrammierung).
- Verarbeitung der im Register bereitgestellten Information.
- Bereitstellung eines neuen Befehls, d.h. es muß dafür sorgen, daß nach Abarbeitung eines Befehls ein neuer zur Dekodierung ins Befehlsregister geladen wird. (Programmzähler mit Adresse des nächsten auszuführenden Befehls laden → Laden des Adressenzwischenspeichers → Einlesen des neuen Befehls vom Speicher ins Datenregister → Laden des Befehlsregisters → Dekodierung des Befehls → Befehlsausführung) (Bild 2.23).

Das Steuerwerk gewährleistet, bezogen auf die Hardware, die Funktionsfähigkeit der Zentraleinheit.

Anschaulich läßt sich das Gesagte mit einem Flußdiagramm zeigen, das die Zustände im Steuerwerk für die Abarbeitung eines Befehls wiedergibt (Bild 2.23). Bevor dieses Flußdiagramm in seiner eigentlichen Bedeutung für die Zentraleinheit erklärt werden kann, bedarf es noch der Erweiterung der bisher betrachteten busorientierten Zentraleinheit auf ein Mikroprozessorsystem oder treffender einen Mikrocomputer.

2.4 Der Mikrocomputer

In diesem Abschnitt werden der Speicher und das Ein-/Ausgaberegister in Verbindung mit der Zentraleinheit, dem Mikroprozessor, vorgestellt. Diese drei Baugruppen bilden zusammen den Mikrocomputer. Bild 2.24 zeigt einen Mikrocomputer, der aus der besprochenen busorientierten Zentraleinheit als Kernstück und dem Speicher sowie den Ein-/Ausgaberegistern besteht. Die beiden letzteren werden behandelt und in einem besonderen Abschnitt wird auf die Verbindungsleitungen zwischen Mikroprozessor und Speicher eingegangen.

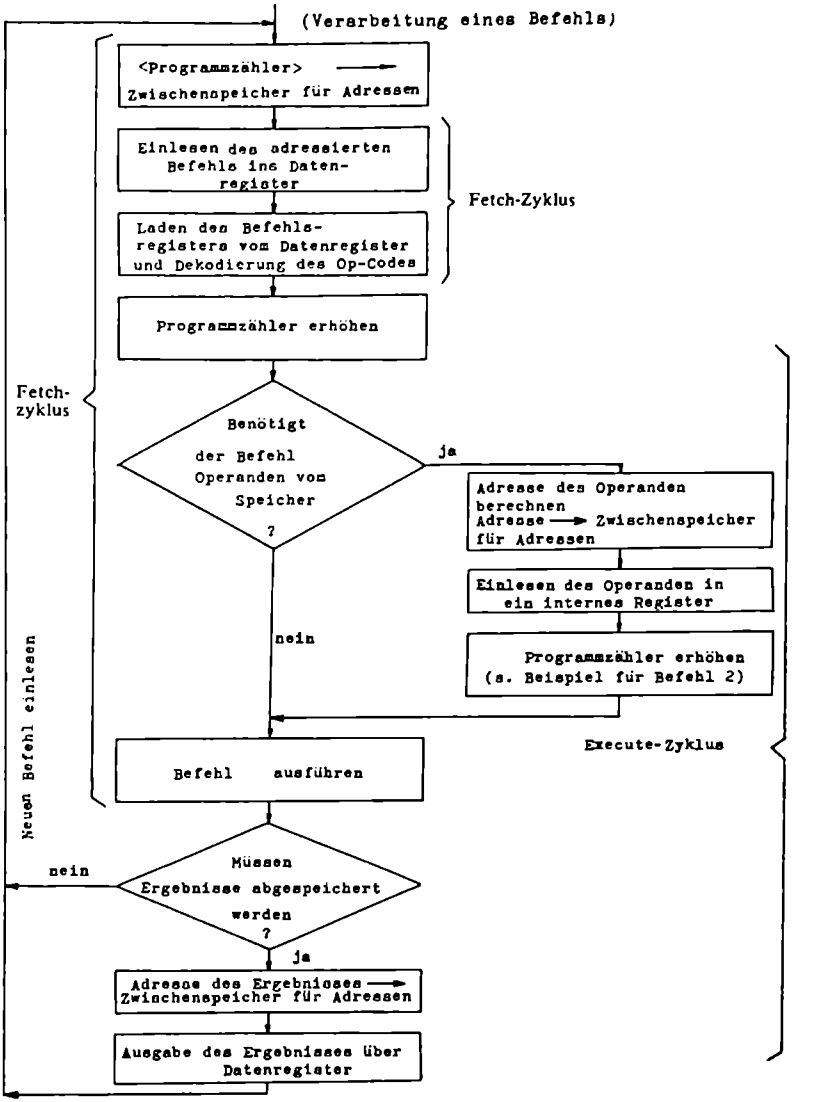


Bild 2.23: Internes Zustandsdiagramm des Steuerwerks

2.4.1 Das Eingabe- und Ausgaberegister

Die Aufgabe dieser Baugruppe läßt sich anhand des Blockschaltbildes (Bild 2.24) ableiten. Sie stellt die Verbindung zwischen Peripherie und Zentraleinheit her, d.h. sie sorgt dafür, daß externen Geräten ein Informationsaustausch mit der Zentraleinheit ermöglicht wird. Die eventuell zusätzlich noch zum Eingabe-/Ausgaberegister benötigten Schaltkreise werden häufig als Anpassungs- oder Interface-Bausteine bezeichnet.

Auf diese Schaltungen wird genauer im Kapitel 5 eingegangen. Sie besitzen unter anderem die Möglichkeit, Steuerinformationen über den Zustand eines externen Gerätes (z.B. Drucker) zu speichern und bei Bedarf an die Zentraleinheit abzugeben.

2.4.2 Der Speicher

Die Steuerung des Mikrocomputers erfolgt über Befehle (Software), die in Form eines Programms im Speicher abgelegt sind. Der Speicher enthält weiterhin Operanden, auf die die Befehle angewandt werden. Wenn die Ausgabe der Ergebnisse nicht an die Peripherie erfolgt, so werden auch diese im Speicher abgelegt.

Zur Kontrolle des Datenaustausches zwischen Speicher und Zentraleinheit stehen, wie in Bild 2.24 gezeigt, Steuerleitungen (R/$\overline{W}$ und Select) sowie zwei Register in der Zentraleinheit zur Verfügung: Zwischenspeicher für Adressen und das Datenregister. Das erste ist für die Ansteuerung des Speichers bzw. eines Speicherplatzes zuständig und das zweite ermöglicht den Austausch von Befehlen und Operanden zwischen Speicher und Zentraleinheit, d.h. ein Programm kann abgearbeitet werden.

Der Adressenzwischenspeicher enthält die Adresse desjenigen Speicherplatzes, in den geschrieben (write) oder von dem ausgelesen werden soll (read). Im Datenregister steht die einzuschreibende Information bzw. wird die ausgelesene bis zur Weiterverarbeitung in der Zentraleinheit zwischengespeichert.

Eine weitere Kenngröße eines Speichers ist seine Kapazität und die interne Organisation der Speicherplätze (Wortlänge).

Da der Speicher aus mehreren kleinen Speichermoduln aufgebaut sein kann, wie z.B. ein 2-k-Speicher mit 8-Bit-Wortlänge aus 16 Bausteinen mit je 1 k × 1 Bit (Bild 2.25), muß die Möglichkeit gegeben sein, diesen Speichermodul anzuwählen. Dies geschieht mittels einer Selectleitung, deren Signal aus den Adreßleitungen abgeleitet wird. Dies ist in Bild 2.24 verdeutlicht.

2.4.3 Speichertypen

In Mikrocomputern werden hauptsächlich zwei gegensätzliche Speichertypen verwendet:

- Schreib-Lese-Speicher,
- Festwert-Speicher.

Der Schreib-Lese-Speicher wird auch als random access memory (RAM)[1]) bezeichnet. Das Kennzeichen dieses Speichers besteht darin, daß Information sowohl abgespeichert als auch ausgelesen werden kann.

Neben den Adreßleitungen besitzt dieser Speicher Datenleitungen zum Abspeichern und Auslesen von Information sowie die beiden Steuerleitungen R/$\overline{W}$ und Select (Bild 2.24).

[1]) Diese Zuordnung hat sich eingebürgert, obwohl sie an sich nichts darüber aussagt, ob der Speicher einen Schreib-Lese- oder Festwert-Speicher darstellt, sondern nur angibt, daß jede Speicherzelle wahlweise adressiert werden kann.

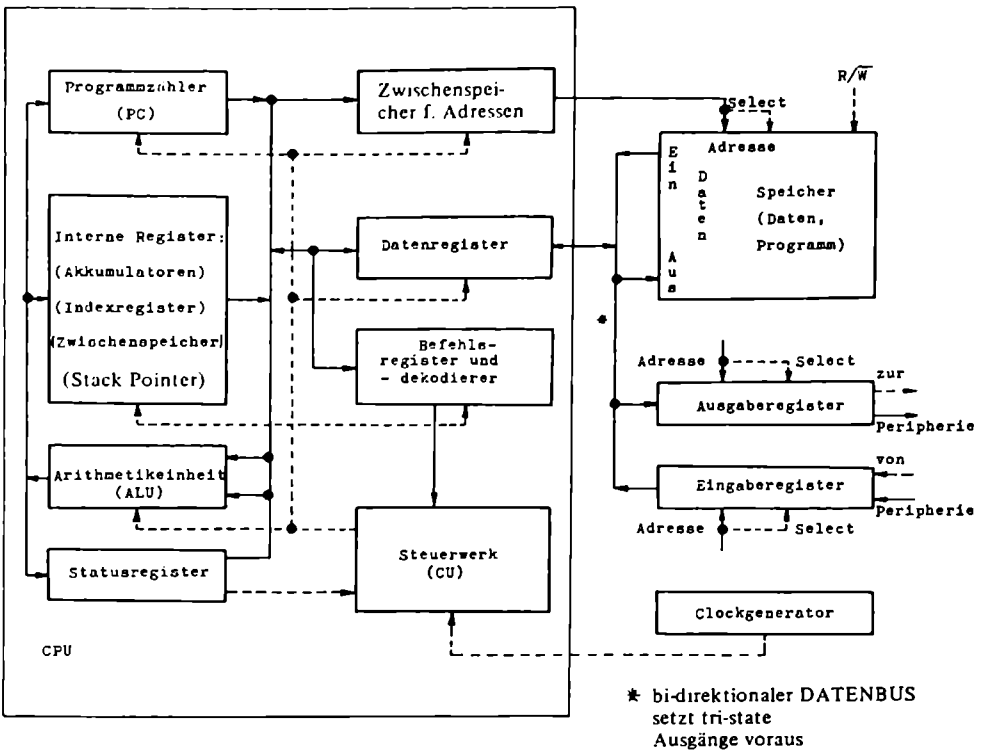


Bild 2.24: Blockschaltbild eines Mikrocomputers

Der Festwert-Speicher, auch als read only memory (ROM) bezeichnet, kann nur ausgelesen werden. Bei diesem Speichertyp reduziert sich somit die Anzahl der Steuerleitungen auf die Selectleitung und zudem sind neben den Adreßleitungen Datenleitungen nur für das Auslesen von Information vorhanden.

Jede der genannten Speichergruppen läßt sich weiter nach anderen Gesichtspunkten unterteilen, nämlich Technologie (Bipolar, MOS), interne Organisation der Speicherzellen (bit- oder wortorganisiert) und bei MOS-RAMs zum Beispiel noch danach, ob die Notwendigkeit einer *periodischen Auffrischung* (Refresh) der einmal abgespeicherten Information notwendig ist oder nicht, d.h. ob der vorliegende Speichertyp *dynamisch* oder statisch ist.

Die Festwert-Speicher werden je nach Herstellungsart in maskenprogrammierte ROMs, d.h. vom Hersteller nach Kundenspezifikation (Bitmuster) gefertigte, und in PROMs (programmable read only memory) bzw. RePROMs (reprogrammable read only memory) unterteilt, d.h. Speicher, die vom Anwender selbst programmiert werden können. PROMs können nur einmal programmiert werden, während dies mit RePROMs mehrmals möglich ist. Zum Beispiel kann ein programmiertes RePROM mit Hilfe von ultraviolettem Licht wieder gelöscht werden.

Ein Vorteil der RePROMs besteht im allgemeinen darin, daß im Gegensatz zum RAM die gespeicherte Information nicht verlorengeht, wenn die Versorgungsspannung abgeschaltet wird. Sie werden aus den beiden letzten Gründen oft zur Programmentwicklung verwendet.

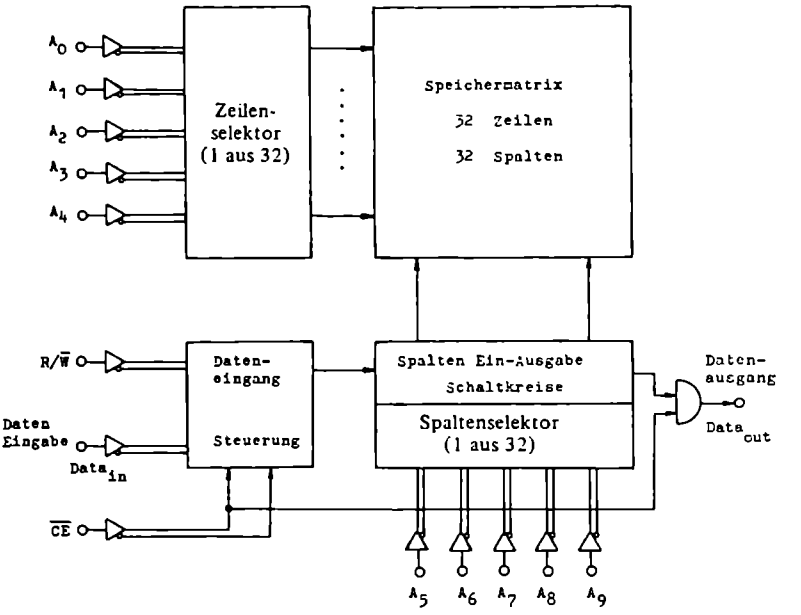


Bild 2.25: 1024 x 1 Bit, statischer Speicher

2.4.4 Verbindung von Mikroprozessor und Speicher

2.4.4.1 Grundlegende Zusammenhänge

Ausgehend von Bild 2.24 wird veranschaulicht, welche Vorgänge in der Zentraleinheit ablaufen, um einen Befehl, der im Speicher abgelegt ist, zu adressieren und auszuführen.

Für diese Betrachtung wird von einer Schaltkreisanordnung ausgegangen, die aus Mikroprozessor (Zentraleinheit (central processing unit = CPU)) und Speicher (memory) besteht. Die Ein-/Ausgaberegister spielen in diesem Zusammenhang keine Rolle und sind deshalb weggelassen (Bild 2.26).

Im Mikroprozessor sind neben dem Steuerwerk noch die Register angegeben, die zur Adressierung eines Speichers und zum Auslesen des Speicherinhalts benötigt werden (Bild 2.26).

Ein Kennzeichen eines Speichers ist seine Kapazität, d.h. wieviele Speicherplätze er umfaßt und wie diese organisiert sind (z.B. 4 k x 1 Bit; 1 k x 8 Bit).

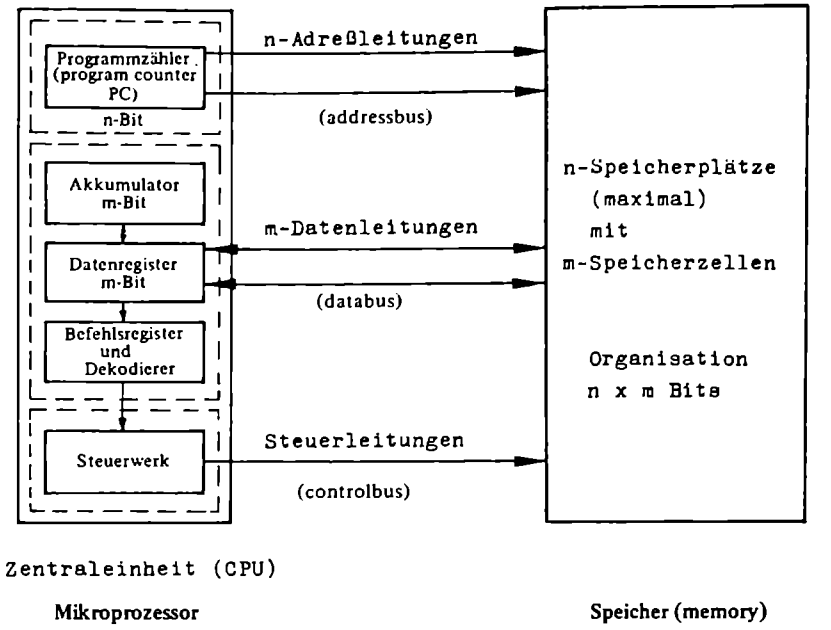


Bild 2.26: Mikroprozessor und Speicher

Bedeutsam sind die folgenden Zuordnungen:

1. Die Anzahl der Adreßleitungen n gibt Auskunft über die Größe des maximal adressierbaren Speicherraumes.
   Es gilt:

max. Speicherraum = 2^n
n = Anzahl der Adreßleitungen

   I

   Demnach können mit 16 Adreßleitungen 64-k Speicherplätze adressiert werden. Die Bezeichnung 64-k ist aus Gründen der Übersichtlichkeit eingeführt worden.

2. Der Speicher wurde als die Baugruppe eingeführt, in der Befehle und Daten abgelegt sind. Diese können zum Mikroprozessor nur über die Datenleitungen gelangen. Da deren Anzahl m für einen Mikroprozessor über die Architektur festgelegt ist, muß ein Speicherwort ebenfalls aus m Speicherzellen (Bits) bestehen. Es muß m-Bit breit sein.

Ein weiterer Gesichtspunkt ist der, daß die Verarbeitung der Operanden in der Register-Arithmetikeinheit erfolgt und daher die im Speicher abgelegten Operanden gleiche Wortbreite besitzen müssen wie die zur Operandenverarbeitung benutzten internen Register, sprich Akkumulatoren.

Es gilt:

$$\boxed{\text{Bitbreite eines Akkumulators} = \text{Anzahl der Datenleitungen} = m} \qquad \text{IIa}$$

und damit

$$\boxed{\text{Anzahl der Bits eines Speicherwortes} = \text{Anzahl der Datenleitungen} = m} \qquad \text{IIb}$$

Die Zuordnungen I und IIa, b verdeutlichen einen wichtigen Zusammenhang zwischen Mikroprozessor-Architektur und Speicherorganisation.

Es gilt:

$$\text{Speicherorganisation} = \text{Speicherraum} \times \text{Anzahl der Datenleitungen},$$

somit

$$\boxed{\text{Speicherorganisation} = 2^n \times m \text{ (maximal)}} \qquad \text{III}$$

Für einen Mikroprozessor, die Zentraleinheit eines Mikrocomputers, der 16 Adreßleitungen und 8 Datenleitungen besitzt, besteht maximal die Möglichkeit, einen Speicher der Organisation 64-k × 8 Bit zu adressieren.

An dieser Stelle sei noch darauf hingewiesen, daß die Voraussetzung getrennter Adreß-, Daten- und Steuerleitungen nur für den Speicher Bedingung ist, während von der Zentraleinheit durchaus Adressen und Daten über dieselben Leitungen übertragen werden können. Jedoch muß in diesem Fall durch zusätzliche Schaltkreise eine Aufspaltung in Adressen und Daten vorgenommen werden, um eine Zusammenarbeit zwischen Zentraleinheit und Speicher zu ermöglichen.

2.4.4.2 Funktionsabläufe beim Einlesen und Verarbeiten eines Befehls

Es sei ein Mikroprozessor mit getrennten Adreß- und Datenleitungen gegeben. Am Beispiel der Ausführung eines kleinen Programms (Bild 2.27) soll das interne Zustandsdiagramm des Steuerwerks erklärt werden. Im folgenden wird auf die Bilder 2.23 und 2.24 Bezug genommen.

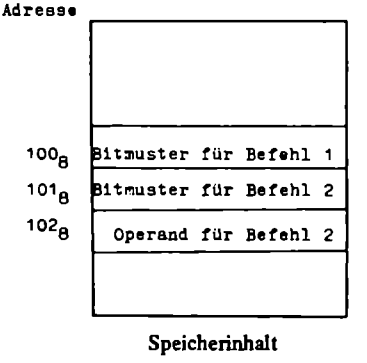


Bild 2.27: Beispiele zur Befehlsverarbeitung

Funktionsabläufe zu Befehl 1

Zur Adressierung des Speichers dient der Programmzähler (program counter = PC). Ein Adressenzwischenspeicher im Mikroprozessor wird mit dem Programmzählerinhalt geladen. Somit steht auf den n Adreßleitungen die Adresse des gewünschten Speicherplatzes, in unserem Beispiel also die Adresse 100_8. Unter dieser Adresse ist das Bitmuster von Befehl 1 binärcodiert abgelegt. Über die m Datenleitungen wird dieses Speicherwort ins Datenregister des Mikroprozessors eingelesen. Da das Steuerwerk aufgrund des internen Zustandsdiagramms weiß, daß es sich bei der ersten Information, die nach der Abarbeitung des vorhergehenden Befehls in das Datenregister eingelesen wird, immer um einen Befehl und nicht um einen Operanden handelt, wird das eingelesene Bitmuster in das Befehlsregister geladen. Dort wird es dekodiert, d.h. alle zu diesem Bitmuster, sprich Befehl, gehörenden internen und externen Vorgänge werden vorbereitet. Diese laufen gesteuert von einem externen Takt in zeitlich richtiger Reihenfolge ab. Benötigt der Befehl keine Operanden vom Speicher, sondern befinden diese sich z.B. bereits in einem der Akkumulatoren, so wird nach vorhergehender Erhöhung des Programmzählerinhalts um Eins der Befehl ausgeführt.

In Bild 2.23 wird der bisher besprochene Teil des Zustandsdiagramms als „Fetch-Zyklus" bezeichnet. Der Mikroprozessor holt (holen engl.: to fetch) das Bitmuster des Befehls ab. Des öfteren wird dieser Zyklus auch als „Op-Code Fetch" bezeichnet, um auszudrücken, daß mit dem eingelesenen Bitmuster, dem Op-Code, dem Mikroprozessor mitgeteilt wird, welchen Befehl er auszuführen hat. Im Fetch-Zyklus kann durchaus die Befehlsausführung (ausführen engl.: to execute) eingeschlossen sein (Bild 2.23). Dieser Fall wurde für Befehl 1 im Beispiel angenommen.

Funktionsabläufe zu Befehl 2

Die Adressierung des Speichers läuft wie oben beschrieben ab (Adresse 101_8) und das Bitmuster von Befehl 2 wird ins Datenregister eingelesen und von dort zum Befehlsregister weitergegeben (Fetch-Zyklus). Das Steuerwerk sorgt für die Erhöhung des Programmzählers und leitet, da Operanden vom Speicher benötigt werden, deren Adressierung in die Wege. Dazu muß die Adresse des Operanden ausgegeben werden. Die Operandenadresse kann Ergebnis einer Adressenberechnung sein oder sie ist, wie im vorliegenden Fall, der um Eins erhöhte Wert des Programmzählerinhalts (102_8). Allgemein nennt man die Zeit, die vergeht, bis die Adresse des Operanden gebildet ist und dieser adressiert werden kann, einen „Defer-Zyklus" (aufschieben engl.: to defer). Liegt die Adresse des Operanden fest, so erfolgt dessen Adressierung. Über das Datenregister wird er eingelesen und entsprechend dem auszuführenden Befehl z.B. zu einem anderen Operanden hinzuaddiert. Anschließend an diesen Rechenschritt wird das Ergebnis mit Hilfe des Statusregisters gekennzeichnet, z.B. ob es gleich Null, negativ oder positiv ist. Das Statusregister hält diese Information als „Status" des Ergebnisses fest und teilt diesen dem Steuerwerk mit.

Des öfteren muß jedoch ein Ergebnis an den Speicher ausgegeben werden. Auch dieser Fall wurde in Bild 2.23 berücksichtigt. Der Speicherplatz für das Ergebnis wird nach erfolgter Adressenberechnung in bekannter Weise adressiert.

Die Zeit zur Ausführung eines Befehls, gerechnet vom Einlesen des Operanden bis zu dessen Verarbeitung, wird als „Execute-Zyklus" bezeichnet. Ein Befehlszyklus setzt sich somit aus den drei Teilen:

Fetch – Defer – Execute

zusammen.

2.4.5 Schreibweise zur allgemeinen Darstellung eines Datenaustausches zwischen Speicher und Zentraleinheit

Der Speicherplatz eines Speichermoduls wird gekennzeichnet durch eine Adresse. Dieser Adresse ist ein ganz bestimmter Inhalt (Speicherwort) zugeordnet. Dieser Sachverhalt wird durch die folgende Schreibweise ausgedrückt:

⟨Adresse A⟩ = X,

d.h. „der Inhalt von" Adresse A wird ausgedrückt durch das Symbol der spitzen Klammern ⟨ ⟩.

Diese Schreibweise wird später noch bei den Adressierungsarten verwendet werden.

Die Anwendung des Klammersymbols ist nicht nur auf Adressen anwendbar, sondern auch auf Register, wie z.B. den Akkumulator oder den Programmzähler.

⟨ACC⟩ = $0000\ 0000_2$

⟨PC⟩ = $0011\ 0001_2$.

Soll nun ein Datenaustausch zwischen einem Akkumulator und dem Speicher dargestellt werden, so muß zuerst noch ein weiteres Symbol eingeführt werden, nämlich das Symbol →.

⟨ACC⟩ → Adresse A

(Ursprungsort = source) (Bestimmungsort = destination)

steht für: Abspeichern oder transferieren des Inhalts von ACC in den Speicherplatz mit der Adresse A.

Das Symbol → bringt somit die Richtung eines Datenaustausches zum Ausdruck.

Zwei häufig bei den Adressierungsarten (Kapitel 3) vorkommende Fälle sollen noch erklärt werden.

1. Fall ⟨ACC⟩ → ⟨Adresse A⟩

steht für: Transferiere den Inhalt des ACC in die Speicherzelle, deren Adresse durch den Inhalt der Adresse A gegeben ist (siehe Kapitel 3 unter indirekter Adressierung).

2. Fall ⟨⟨Adresse A⟩⟩ → ACC

steht somit für: Transferiere den Inhalt der Speicherzelle, deren Adresse durch den Inhalt der Adresse A gegeben ist, in den ACC (siehe ebenfalls Kapitel 3 unter indirekter Adressierung).

Einige Beispiele sollen das Gesagte verdeutlichen.

Speicher

	Adresse				Inhalt			
	A_3	A_2	A_1	A_0				
0	0	0	0	0	0	1	0	1
1	0	0	0	1	0	0	0	0
2	0	0	1	0	0	1	1	1
3	0	0	1	1	0	1	0	1
4	0	1	0	0	0	1	1	0
5	0	1	0	1	1	0	1	1
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	1	1	0
9	1	0	0	1	1	1	0	1
A	1	0	1	0	1	1	0	0
B	1	0	1	1	1	0	1	1
C	1	1	0	0	1	1	1	1
D	1	1	0	1	0	0	0	0
E	1	1	1	0	1	1	0	1
F	1	1	1	1	0	0	0	1

Befehl	Ergebnis
⟨C⟩ → ACC	⟨ACC⟩ = 1111_2
⟪E⟫ → ACC	⟨ACC⟩ = 0000_2
⟨9⟩ →⟨ACC⟩ mit ⟨ACC⟩ = 0101	⟨5⟩ = 1101_2
⟨A⟩ → ACC	⟨ACC⟩ = 1100_2

Man merke sich: Ein Register wird, wenn es Ursprungsort eines Datenaustausches ist, immer mit Klammersymbol geschrieben (z.B. ⟨ACC⟩ →).

Kapitel 3
Überblick über Adressierungsarten

3.1 Problemstellung

Nachdem bisher die Baugruppen des Mikrocomputers behandelt wurden, sind die beiden nächsten Kapitel einem weiteren wesentlichen Aspekt gewidmet, nämlich den Adressierungsarten und dem Befehlsvorrat. Auf die Betrachtung der Hardware folgt die der mikroprozessorspezifischen Software. In diesem Kapitel werden die gebräuchlichsten Adressierungsarten vorgestellt und ihr Einsatz sowie ihre Leistungsfähigkeit aufgezeigt.

Der Mikroprozessor ist auf den Speicher angewiesen, weil dort unter anderem die Befehle abgelegt sind, die er abarbeiten muß, um eine spezielle Aufgabe zu lösen. Viele Befehle benötigen Operanden, die entsprechend der im Befehl festgelegten Funktion verknüpft werden. Außerdem müssen Ergebnisse für eine spätere Ausgabe im Speicher abgelegt werden. Den Befehlen und Daten (Operanden, Ergebnisse) sind im Speicher eindeutig Speicherplätze – mit anderen Worten Adressen – zugeordnet, an denen sie abgelegt bzw. abzulegen sind. Mit dem Begriff „Adressen" ist bereits das Stichwort für dieses Kapitel gefallen. Im folgenden geht es darum, die Problematik aufzuzeigen, die bei der Adressierung von Daten und Befehlen im Speicher besteht.

Es wird wieder der Transfer Mikroprozessor – Speicher von Bedeutung sein. Die Verbindung zwischen Speicher und Mikroprozessor wird durch Adreßleitungen, Datenleitungen und Steuerleitungen vorgenommen. Die Anzahl der Adreßleitungen bestimmt den maximal adressierbaren Speicherraum, die der Datenleitungen legt ihrerseits die Speicherwortbreite fest. Mit der Speicherwortbreite ist das Befehls- und Datenformat in seiner Breite vorgegeben. Für einen 8-Bit-Mikroprozessor, d.h. 8-Bit-Datenformat, hat dies zur Folge, daß zur Charakterisierung eines Befehls ebenfalls im ganzen 8 Bits zur Verfügung stehen. Der Mikroprozessor leitet aus einem über die Datenleitungen eingelesenen Befehl durch Dekodierung des Operationscodes (Op-Code) die auszuführenden Funktionen ab. Weiterhin entnimmt er aus dem Op-Code Information darüber, an welcher Adresse der benötigte Operand abgelegt ist oder wie dessen Adresse berechnet werden kann. Der Mikroprozessor muß im letzten Fall eine Adressenberechnung durchführen.

Aus dem bisher Gesagten läßt sich folgendes Fazit ziehen: Das Bitmuster eines Befehls muß sowohl Information über dessen Funktion als auch über die Adresse des Operanden enthalten. Daher sind Verfahren notwendig, die es trotz der nur in beschränktem Maße zur Verfügung stehenden Datenleitungen gestatten, Operanden zu adressieren, die irgendwo im definierten Speicherraum oder im Registerbereich des Mikroprozessors abgelegt sind. Die Verfahren werden Adressierungsarten genannt.

Für einen 8-Bit-Mikroprozessor mit 16 Adreßleitungen (8-Bit-Datenbus, 16-Bit-Adreßbus) muß es möglich sein, 2^{16}, also 64 k, Speicherplätze zu adressieren, und zwar im Rahmen des 8-Bit-Befehlsformats. In bezug auf den erwähnten internen Registerbereich kann an dieser Stelle bemerkt werden, daß die Adressierung eines Registers dieses Bereiches ungleich weniger Schwierigkeiten für die Aufteilung innerhalb des Befehlsformats macht, da die Registerzahl meistens unter 64 liegt. Daher sind meistens 5 Bits zur Kennzeichnung ausreichend.

Durch die Anwendung der Adressierungsarten wird es möglich, den gesamten definierten Speicherraum zu adressieren. Andere Gesichtspunkte, die Gründe für die Vielfalt der Adressierungsarten angeben, werden kurz dargelegt:

1. Schnelle Adressierung eines Operanden.
2. Speicherplatzsparende Anordnung der Adresse des Operanden.
3. Verarbeitung von Operanden, die in ihrer Anordnung einer Struktur unterliegen, z.B. Listen, Tabellen.
4. Vereinfachung von Programmen (z.B. Unterprogramme, siehe Beispiele Kapitel 4).

Nachdem die Problematik eingehend dargestellt wurde, werden Beispiele gegeben, die verdeutlichen sollen, welche Möglichkeiten bestehen, Adressierungsarten bei Mikroprozessoren zu verwirklichen. Ziel dieser Betrachtung ist es, einen Einblick zu geben, wie in einem vom Befehlsformat her eingeschränkten Rahmen verschiedene Adressierungsarten denkbar sind. Dies soll anhand dreier verschiedener Befehlsformate geschehen. Anschließend werden Beispiele verschiedenster Adressierungsarten besprochen.

Befehlsformat 1

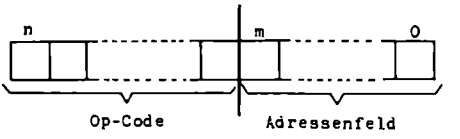


Bild 3.1: Aufteilung in Op-Code und Adressenfeld

Für einen Mikroprozessor mit einem (n + 1)-Bit-Datenbus hat die Aufteilung in Op-Code und Adressenfeld zur Folge, daß (n–m) Bits für den Op-Code und (m + 1) Bits für das Adressenfeld zur Verfügung stehen. Diese Einteilung wäre für einen 16-Bit-Mikroprozessor – d.h. 16-Bit-Datenformat – durchaus denkbar, da z.B. bei einem 4-Bit-Op-Code ein 12-Bit-Adressenfeld vorhanden wäre. Damit ließe sich ein Speicherraum von 2^{12} = 4 k Speicherplätzen über das Adressenfeld direkt adressieren. Bild 3.1 verdeutlicht auch, daß für Datenformate, wie z.B. 8 Bit, dieses Befehlsformat untauglich ist. Werden von den 8 Bits 3 für den Op-Code verwendet, so stehen die restlichen 5 noch für das Adressenfeld zur Verfügung. Die auf diese Weise im Adressenfeld festlegbaren Speicherplätze sind auf 32 beschränkt. Dies ist für eine praktische Anwendung völlig unzureichend.

Befehlsformat 2

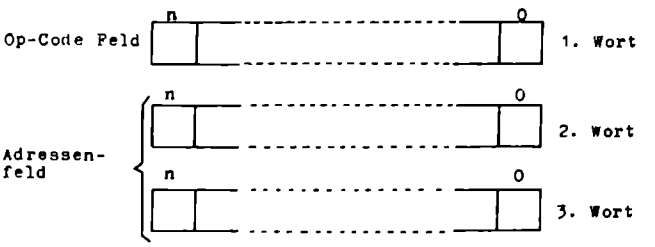


Bild 3.2: Drei-Wort-Befehl (ein Wort Op-Code, zwei Worte Adressenfeld)

Bild 3.2 zeigt, daß durch Erweiterung des Adressenfeldes auf zwei Worte der adressierbare Speicherraum gegenüber Bild 3.1 merklich ausgedehnt werden kann.

Dieses Befehlsformat wird am Beispiel eines 8-Bit-Mikroprozessors näher erklärt. In Bild 3.2 ist n = 7 und anstelle der Bezeichnung Wort wird der Begriff „Byte" eingeführt. 8 Bits werden als 1 Byte bezeichnet. Durch die erfolgte Aufteilung in 1 Byte für Op-Code und 2 Bytes für das Adressenfeld (2 Bytes = 16 Bits) wird der adressierbare Speicherraum auf 64 k Speicherplätze ausgedehnt. Befehle, die dieses Format besitzen, werden je nach Anzahl der Bytes Zwei-Byte- oder Drei-Byte-Befehle genannt. Ein Nachteil dieser Methode ist, daß derartige Befehle bis zu drei Speicherplätze zur Darstellung benötigen und ihre Verarbeitung deshalb auch mehr Zeit erfordert. Der Grund ist darin zu sehen, daß zur Verarbeitung des Befehls der Speicher bis zu dreimal hintereinander adressiert werden muß, um die gesamte Information über Befehl *und* Adresse des Operanden in den Mikroprozessor zu erhalten, um anschließend den Operanden zu adressieren, einzulesen und zu verarbeiten.

Befehlsformat 3a

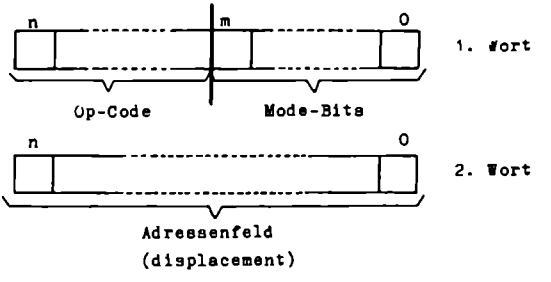


Bild 3.3a: Zwei-Wort-Befehl mit zusätzlichen Mode-Bits zur Festlegung des Basisregisters (**Spezialfall:** Ein-Wort-Befehl ohne Adressenfeld)

Befehlsformat 3b

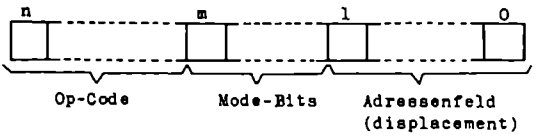


Bild 3.3b: Ein-Wort-Befehl unterteilt in Op-Code, Mode-Bits und Adressenfeld

In einem dritten Befehlsformat wird der Nachteil des vorhergehenden dadurch umgangen, daß wie in Bild 3.1 ein Ein-Wort- bzw. wie in Bild 3.2 ein Zwei-Wort-Befehl verwendet wird, jedoch mit der Neuerung, daß mit Hilfe von zusätzlichen Bits (Mode-Bits) festgelegt wird, wie der Mikroprozessor die Adresse des Operanden bildet (= Effektive Adresse (effective address = EA) des Operanden).

Über diese Mode-Bits (Art und Weise engl.: mode) wird ein Basisregister ausgewählt, das zusammen mit dem Adressenfeld zur Adressierung des Operanden herangezogen wird. Der Begriff „Basisregister" beinhaltet, daß die Adresse des Operanden nicht wie bei den beiden vorhergehenden Befehlsformaten im Speicher steht und damit Bestandteil des Befehls ist, sondern daß das Adressenfeld und das Basisregister zur Berechnung der Operandenadresse herangezogen werden (Adressenberechnung). Das Adressenfeld wird in diesem Zusammenhang als Distanzadresse (displacement (disp.), offset) bezeichnet. Sie wird häufig als Dualzahl in Zweierkomplementdarstellung aufgefaßt.

Zusammenfassend läßt sich sagen, daß durch die verfügbaren Adressierungsarten Lösungsmöglichkeiten angeboten sind, jeden Speicherplatz zu adressieren, und zwar im Rahmen der für einen Mikroprozessor definierten Befehlsformate. Die ausgewählten Formate hängen von verschiedenen Faktoren ab, die die Mikroprozessorhersteller im Auge haben, wie zum Beispiel:

- Architektur (→ Datenbusbreite)
- Anzahl der möglichen Befehle (→ Breite des Op-Code-Feldes)
- gewünschter Speicherraum (→ Adreßbusbreite)

Bei Mikroprozessoren mit 4- und 8-Bit-Datenformat findet man häufig die Befehlsformate 2 und 3a, bei solchen mit 16-Bit-Datenformat dagegen meist die Formate 1 und 3b.

3.2 Allgemeine Einteilung der Adressierungsarten

Da in der von Mikroprozessorherstellern angebotenen Literatur eine ziemlich wahllose Benennung der Adressierungsarten anzutreffen ist, soll in diesem Abschnitt eine Einteilung vorgeschlagen werden, die es erlaubt, jede Adressierungsart einer der in Bild 3.4 benannten Gruppen zuzuordnen. Bei der hier vorgeschlagenen Einteilung wurde von der folgenden Überlegung ausgegangen:

Alle Adressierungsarten lassen sich einteilen im Hinblick auf das Basisregister, das zur Berechnung der effektiven Adresse (EA) des Operanden verwendet wird. Die Einteilung sieht eine Gruppe „Programmzähleradressierungsarten" und eine Gruppe „Registeradressierungsarten" vor. Das Basisregister ist demnach zum einen der Programmzähler und zum anderen ein internes Register des Mikroprozessors (z.B. Akkumulator, Indexregister, Stack Pointer).

Bevor zu den einzelnen Adressierungsarten übergegangen wird, sei noch auf eine häufig anzutreffende Schreibweise hingewiesen, die eine übersichtliche Darstellung der einzelnen Adressierungsarten gestattet. Der Inhalt des Programmzählers (⟨PC⟩) gibt bekanntlich Auskunft über die Adresse des gerade ausgeführten Befehls.

Speicheradresse	Speicherinhalt
⟨PC⟩ = N	Befehl
N + 1	Operand

Soll der unter der Speicheradresse N abgelegte Befehl ausgeführt werden, so muß der Inhalt des Programmzählers (⟨PC⟩) gleich N sein. Um nicht jedesmal die vollständige Adresse ausschreiben zu müssen, wird für die Adresse N des gerade ausgeführten Befehls das Symbol „ . " (Punkt engl.: dot) eingeführt.

Obiges Beispiel läßt sich mit dem Symbol „ . " folgendermaßen darstellen:

Speicheradresse	Speicherinhalt
⟨PC⟩ = .	Befehl
. + 1	Operand

Dot (.) steht demnach für N, „. + 1" für N + 1. Das Symbol „." wird im weiteren Verlauf des Kapitels häufig verwendet.

In den weiteren Ausführungen werden die in Bild 3.4 zusammengestellten Adressierungsarten besprochen. Dabei wird so vorgegangen, daß zuerst die Gruppe der direkten Adressierungsarten und anschließend die der indirekten Adressierungsarten behandelt wird. Dies gilt sowohl für die Register- als auch für die Programmzähleradressierungsarten. Für jede Adressierungsart ist ein Beispiel und ein Symbol angegeben, um ihre Funktion, besonders die notwendige Adressenberechnung, zu verdeutlichen. Durch den Hinweis auf das Befehlsformat kann der Befehl einem der beschriebenen Formate zugeordnet werden. Dies gibt Aufschluß über die Geschwindigkeit eines Befehls und seine Verwendungsmöglichkeit, um Operanden zu adressieren. Zu der Darstellung der Beispiele sind noch einige erklärende Vorbemerkungen erforderlich.

Als Demonstrationsobjekt dient ein 8-Bit-Mikroprozessor mit 16 Adreßleitungen und einem 8-Bit-Adreßregister RO, welches neben dem Indexregister und dem Programmzähler, die beide 16-Bit-Register sind, als Basisregister verwendet wird. Dabei wurde bewußt in Kauf genommen, daß die Wahl eines 8-Bit-Adreßregisters RO den adressierbaren Speicherraum auf $2^8 = 256$ Speicherplätze begrenzt. Das Ziel der Beispiele ist es in erster Linie, zu veranschaulichen, wie die Adressenberechnung durchgeführt wird, um den gewünschten Operanden im Speicher oder in einem internen Register zu adressieren.

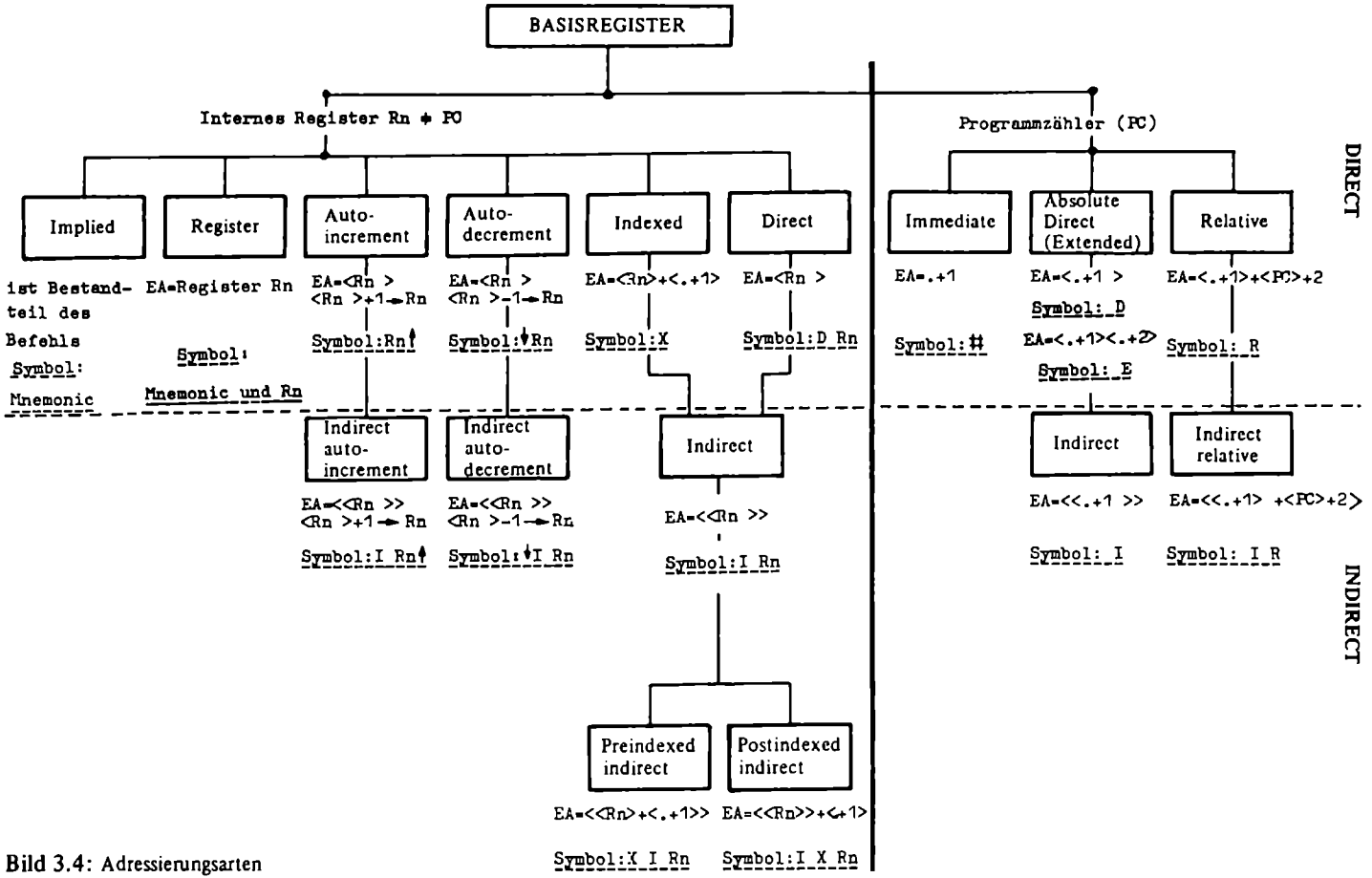


Bild 3.4: Adressierungsarten

In den Beispielen werden einige Bezeichnungen immer wieder vorkommen, die im folgenden erläutert werden:

N	Adresse des gerade verarbeiteten Befehls,
⟨PC⟩	Inhalt des Programmzählers (PC),
EA	Effektive Adresse des Operanden,
/	Hinter diesem Schrägstrich steht die Erklärung des Befehls in Klammerschreibweise.

Die Bilder, an denen die Adressierungsarten veranschaulicht werden, sind so aufgebaut, daß deutlich wird, wie der Mikroprozessor (CPU) und der Speicher an der Bildung der effektiven Adresse (EA) mitwirken.

3.3 Direkte Adressierung

Definition

Unter *direkter* Adressierung wird verstanden, daß zur Bildung der effektiven Adresse des Operanden (Daten, Befehl) *höchstens* eine Adressenberechnung notwendig ist.

Aus den verschiedenen Möglichkeiten der Adressenberechnung ergeben sich die im folgenden vorgestellten direkten Adressierungsarten, unterteilt in Register- und Programmzähleradressierungsarten.

3.4 Direkte Registeradressierungsarten

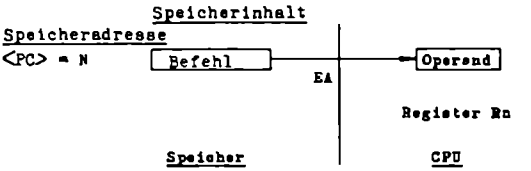
3.4.1 Implied

Wird ein Befehl mit dieser Adressierungsart verwendet, so bedeutet dies, daß aus dem Operationscode und den Mode-Bits hervorgeht, welcher Operand eines Registers verarbeitet werden soll. Häufig wird diese Adressierungsart auch als inherent bezeichnet, was soviel bedeutet wie „der Ort des Operanden ist Bestandteil des Befehls" (angeboren, eigen engl.: inherent).

Ein Befehl, der *implied* Adressierung verwendet, wäre zum Beispiel:

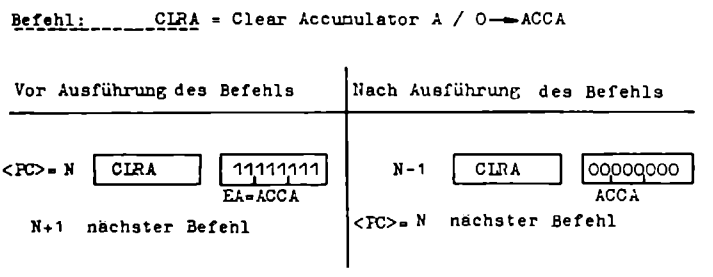
Clear Accumulator (CLRA)

Aus diesem Befehl geht hervor, daß der Akkumulator gelöscht werden soll. Die effektive Adresse ist bekannt, nämlich der Akkumulator, sie muß nicht gesondert berechnet werden.



EA ist Bestandteil des Befehls
Symbol: jeweiliger Name des Befehls (z.B. CLRA)
Befehlsformat: 3a (Spezialfall)

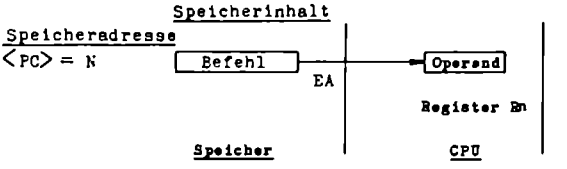
Beispiel zur Adressierungsart: Implied



3.4.2 Register

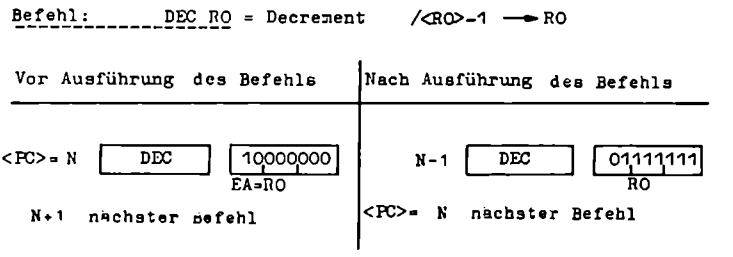
Bei dieser Adressierungsart, die vom Aufbau her der vorhergehenden sehr ähnlich ist, wird zusätzlich zum Befehlsnamen (Mnemonic) noch das Register benannt, auf das der Befehl angewandt wird. Das Register enthält den Operanden.

Die Adressierungsart *Register* wird vorteilhaft dort verwendet, wo für Rechenoperationen häufig das gleiche Register verwendet wird (z.B. Erniedrigen eines Adreßregisters). Diese Befehle werden schnell ausgeführt, da der Operand sich in einem internen Register befindet.



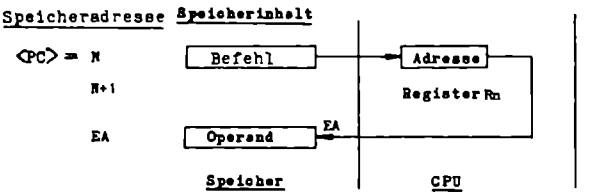
EA = Rn
Symbol = jeweiliges Mnemonic sowie Register Rn (DEC R0)
Befehlsformat: 3a (Spezialfall)

Beispiel zur Adressierungsart: Register



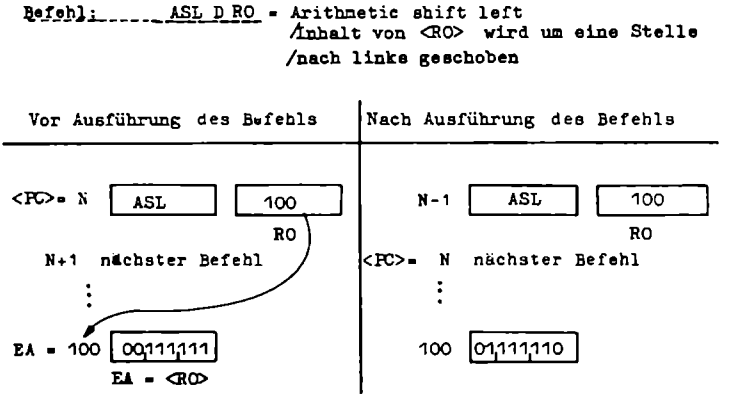
3.4.3 Register direct

Bei dieser Adressierungsart wird die effektive Adresse des Operanden durch den Inhalt eines Registers bestimmt. Der Operand ist im Speicher abgelegt und muß von dort eingelesen werden. Ein Befehl mit dieser Adressierungsart erlaubt die Adressierung von Operanden in einem Speicherraum, der der Wortbreite des ausgewählten Registers entspricht. Die Ausführungszeit des Befehls ist länger, als wenn der Operand in einem internen Register abgelegt ist, weil der Speicher zum Einlesen des Operanden adressiert werden muß.



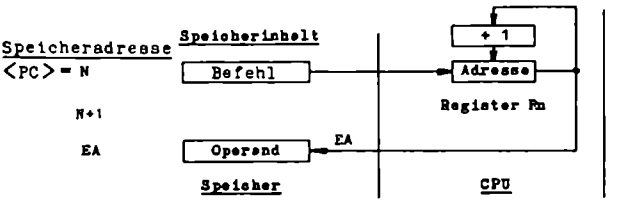
EA = ⟨Rn⟩
Symbol: D Rn
Befehlsformat: 3a (Spezialfall)

Beispiel zur Adressierungsart: Register direct



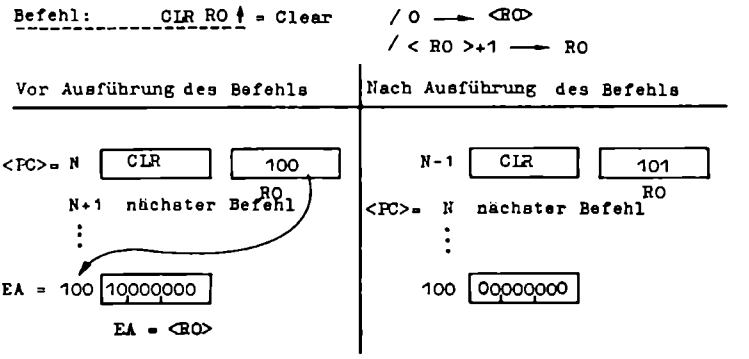
3.4.4 Autoincrement

Unter *autoincrement* versteht man eine Adressierungsart, bei der der Inhalt des im Befehl spezifizierten Registers als Adresse des Operanden (EA) verstanden wird. *Nach* ausgeführtem Befehl wird der Registerinhalt automatisch um Eins erhöht (increment). Eine andere Bezeichnung nennt diese Adressierungsart deshalb auch *postincrement*. Dieser Adressierungsart bedient man sich vorteilhaft bei der Verarbeitung von Listen und Tabellen sowie beim Transfer von Argumenten für Unterroutinen (siehe Beispiel Kapitel 4).



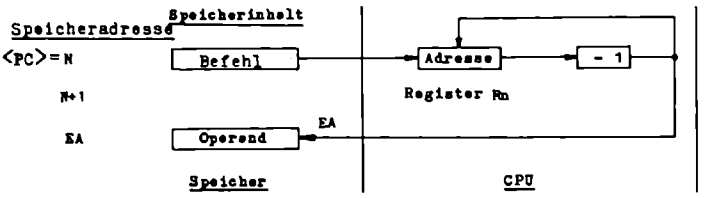
EA = ⟨Rn⟩
Nach Ausführung des Befehls: ⟨Rn⟩ + 1 → Rn
Symbol: Rn ↑
Befehlsformat: 3a (Spezialfall)

Beispiel zur Adressierungsart: Autoincrement



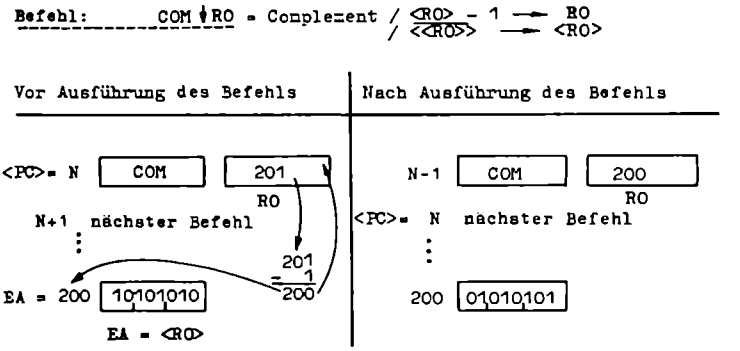
3.4.5 Autodecrement

Diese Adressierungsart wird wie die vorhergehende gebildet, mit dem einzigen Unterschied, daß der Inhalt des Registers, das die Adresse des Operanden enthält, um Eins erniedrigt wird (decrement), und zwar *vor* der Ausführung des Befehls. Deshalb wird diese Adressierungsart auch des öfteren als *predecrement* bezeichnet. Diese Adressierungsart wird bei der Verarbeitung von Listen und Tabellen benutzt (s. autoincrement).



EA = ⟨ Rn ⟩
Vor Ausführung des Befehls : ⟨ Rn ⟩ − 1 → Rn
Symbol : ↓ Rn
Befehlsformat : 3a (Spezialfall)

Beispiel zur Adressierungsart: Autodecrement



Bei den Adressierungsarten *autoincrement* und *autodecrement* war die automatische Veränderung des Registerinhalts vor (pre-) oder nach (post-) der Befehlsausführung von Bedeutung. Wesentlich ist in diesem Zusammenhang, daß immer das Paar „pre-, post-" vorkommt. Es sind die beiden folgenden Zuordnungen von pre- und post- zu autoincrement und autodecrement üblich:

1. preincrement und postdecrement,
   d.h. *vor* Befehlsausführung wird der Registerinhalt um Eins erhöht (Symbol ↑ Rn) und *nach* Befehlsausführung um Eins erniedrigt (Symbol Rn ↓).

2. postincrement und predecrement,
   d.h. *nach* Befehlsausführung wird der Registerinhalt um Eins erhöht (Symbol Rn ↑)
   und *vor* Befehlsausführung wird der Registerinhalt um Eins erniedrigt (Symbol ↓ Rn).

Die beiden verschiedenen Formen der Adressierungsarten *autoincrement* und *autodecrement* ermöglichen den Zugriff zu Operanden, die nacheinander abgearbeitet werden sollen (siehe Beispiel Kapitel 4). Weiterhin können sehr einfach benachbarte Operanden verarbeitet werden, da diese Adressierungsarten eine automatische Erhöhung bzw. Erniedrigung des Registerinhalts, d.h. der Adresse des Operanden, bedingen.

3.4.6 Register indexed

Die Adressierungsart *indexed* wird in Mikroprozessoren sehr häufig angetroffen. Sie bedingt bei 8-Bit-Mikroprozessoren einen 2-Byte-Befehl und meistens einen 1-Wort-Befehl bei 16-Bit-Mikroprozessoren.

Andere Namen für diese Adressierungsart sind: *Register relative* oder *Register indirect*. Es soll sich jeweils um *indexed* Adressierung handeln, wenn zur Bildung der effektiven Adresse (EA) für diese Adressierungsart ein Register, das vom Programmzähler (PC) *verschieden* ist, benutzt wird.

Dieses Register wird oft auch als Indexregister bezeichnet.

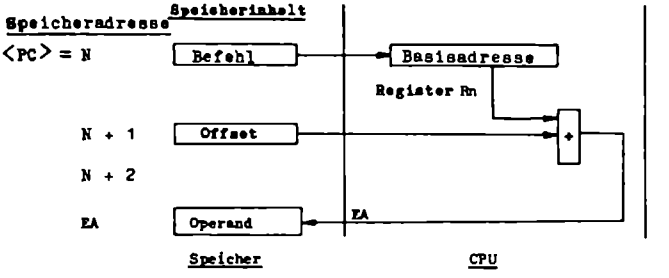
Es kann per Programm geladen werden, kann inkrementiert und dekrementiert werden. Seine Wortlänge entspricht häufig der Anzahl der Adreßleitungen, so daß mit dieser Adressierungsart der gesamte Speicherraum adressiert werden kann.

Diese Adressierungsart stellt eine große Erleichterung für den Programmierer dar, da mit ihrer Hilfe sehr leicht ein *wahlfreier* Zugriff zu Listen realisiert werden kann. Die effektive Adresse des Operanden errechnet sich als die Summe aus Basisadresse ((Rn)) und Offset bzw. Displacement. Der Offset wird häufig als Zweierkomplementzahl interpretiert. Dadurch ist es möglich, bei fester Basisadresse durch Wahl des Offset einen Speicherraum wahlfrei zu adressieren, der z.B. im Falle eines 8-Bit-Datenformats im Bereich

$$\text{Basisadresse} - 128 \leqslant \text{Speicherraum} \leqslant \text{Basisadresse} + 127$$

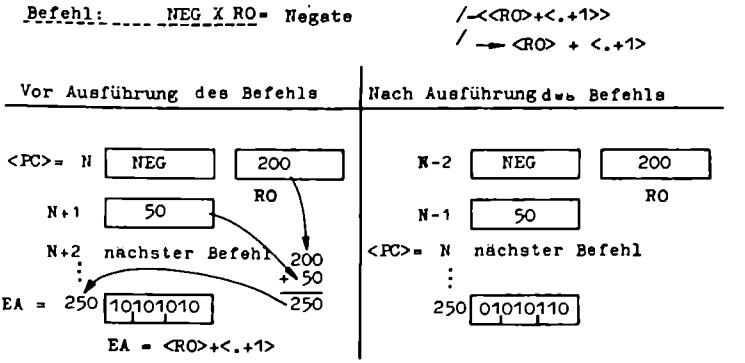
liegen kann.

Diese Adressierungsart gehört wie die indirekte Adressierung zur Reihe derer, die die Adresse unter Programmkontrolle errechnen (computed addressing).



EA = (Rn) + (.+1)
Symbol: X
Befehlsformat: 3a oder 3b

Beispiel zur Adressierungsart: Register indexed



3.5 Direkte Programmzähleradressierungsarten

Die Adressierungsarten, die mit dem Programmzähler als Basisregister möglich sind, sind mit den Registeradressierungsarten vergleichbar. Jedoch haben sich im Laufe der Zeit andere Namen eingebürgert. Im folgenden soll deshalb auch neben den jeweiligen Adressierungsarten der Zusammenhang zwischen Register- und Programmzähleradressierungsarten aufgezeigt werden.

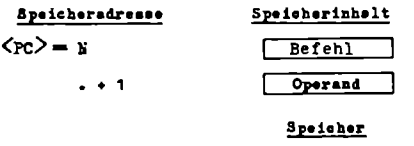
3.5.1 Immediate

Diese Adressierungsart wird vorteilhaft angewandt, um z.B. ein Indexregister oder den Akkumulator zu laden, da der Operand an einem dem Befehl folgenden Speicherplatz abgelegt ist. Eine Adressenberechnung ist nicht notwendig, da der Programmzähler automatisch um Eins erhöht wird, nachdem ein Befehl eingelesen und dekodiert wurde. Der Programmzähler enthält damit bereits die Adresse des Operanden. Befehle, die diese Adressierungsart besitzen, gestatten es, interne Register schnell mit einem Operanden zu laden. Geschieht dieser Vorgang zu Beginn eines Programms, so spricht man von der „Initialisierung" eines Registers.

Als Beispiel sei ein Ladebefehl gegeben:

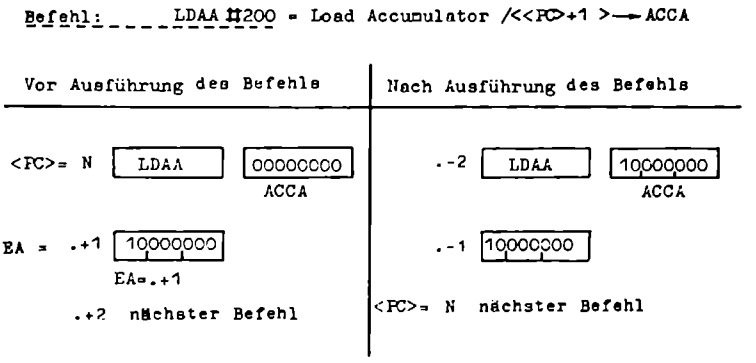
Load Accumulator A immediate mit 3
L D A A # 3

Diese Adressierungsart entspricht der Adressierungsart *autoincrement*, wenn nicht der Programmzähler (PC) als Basisregister ausgewählt ist.



EA = . + 1
Symbol: #
Befehlsformat: 2

Beispiel zur Adressierungsart: Immediate

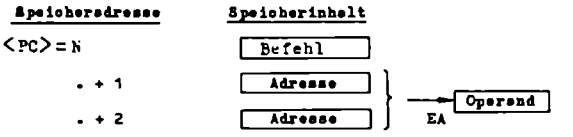


3.5.2 Absolute oder direct

Mit dieser Adressierungsart kann ein Operand an jeder Stelle im Speicher adressiert werden, vorausgesetzt, daß für die Adresse genügend Bytes vorgesehen werden. Für einen 8-Bit-Mikroprozessor mit einem 16 Bit breiten Adreßbus bedeutet dies, daß 2 Bytes für die Adresse vorgesehen werden müssen, um den gesamten Speicherraum zu adressieren.

Vergleicht man die Adressierungsart *absolute* mit der Adressierungsart *Register indexed* (3.4.6), so ergibt sich, daß beide 2-Wort-Befehle bedingen. Der Unterschied beider Adressierungsarten ist in der Größe des adressierbaren Speicherraumes zu sehen. Die *absolute* gestattet, wenn ein 8-Bit-Mikroprozessor vorausgesetzt wird, die Adressierung von 256 Speicherplätzen, während bei *Register indexed* die Möglichkeit gegeben ist, den gesamten Speicherraum zu adressieren, sofern das Indexregister in seiner Registerbreite der des Programmzählers entspricht.

Im Zusammenhang mit der Adressierungsart *absolute* sind noch einige Begriffe zu klären. Ein 8-Bit-Mikroprozessor mit 16 Adreßleitungen kann 64 k Speicherplätze adressieren. Aufgrund des 8-Bit-Datenformats wird der Speicherbereich in Seiten (pages) eingeteilt. Die oberen 8 Bits des Programmzählers dienen als page-Adresse, während die unteren 8 Bits die laufende Adresse der page bestimmen. Eine page umfaßt in diesem Fall $2^8 = 256$ Speicherplätze. Man spricht in diesem Zusammenhang von *absolute zero page addressing*, wenn die page-Adresse den Wert *Null* besitzt. Diese Adressierungsart wird in vielen 8-Bit-Mikroprozessoren angewandt, da sie schnell ist und nur 2 Bytes (1 Byte Op-Code, 1 Byte für Adresse) benötigt werden. Sie wird des öfteren auch als *direct* bzw. *absolute* Adressierung bezeichnet im Gegensatz zur *extended* Adressierung, die 3 Bytes benötigt (1 Byte Op-Code, 2 Bytes für Adresse).

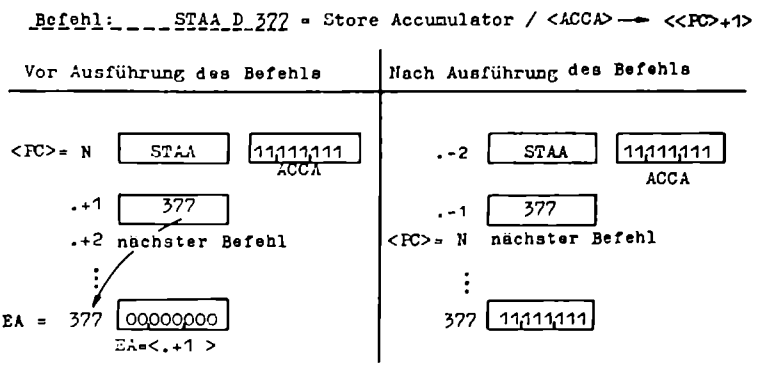


EA = ⟨. + 1⟩ (absolute) bzw. ⟨. + 1⟩ ⟨. + 2⟩ (extended)

Symbol: D Symbol: E

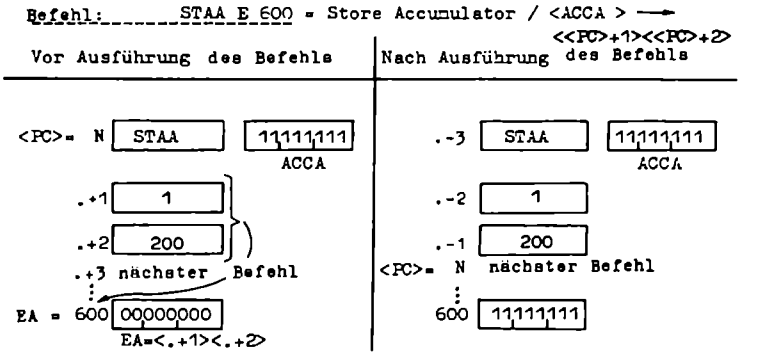
Befehlsformat: 2

Beispiel zur Adressierungsart: Absolute (direct)



Anmerkung: Die höchste Adresse, die mit 8 Bits dargestellt werden kann, ist 11 111 111 = 377_8.

Beispiel zur Adressierungsart: Extended



Anmerkung: Die höchste Adresse, die mit 16 Bits dargestellt werden kann, ist 1 111 111 111 111 111 = 177777_8

im Beispiel: high bits low bits
0 000 000 110 000 000 = 600_8
<.+1> <.+2>

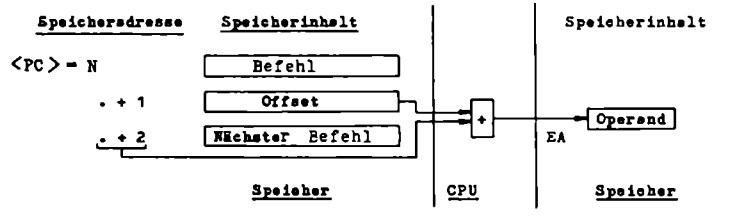
3.5.3 Relative

Diese Adressierungsart gleicht der Adressierungsart *Register indexed*. Jedoch muß ein grundsätzlicher Unterschied zwischen beiden Adressierungsarten erwähnt werden, nämlich bezüglich des Offset bzw. Displacement. Bei *Register indexed* kann der Anwender, d.h. der Programmierer, den Offset nach eigenem Gutdünken einsetzen. Er kann ein im Rahmen des Datenformats mögliches Zweierkomplement einsetzen. Anders verhält es sich mit der Adressierungsart *relative*. Dies hängt mit der Anwendung dieser Adressierungsart zusammen. Sie findet vornehmlich bei Befehlen Verwendung, die Sprünge (jumps) innerhalb eines Programms ausführen. Dazu muß ein Sprungziel, d.h. eine Adresse, bekannt sein, an die gesprungen werden kann. Der Offset stellt nun die Differenz zwischen Sprungziel und augenblicklichem Programmzählerstand dar. Mit anderen Worten: Für einen 8-Bit-Mikroprozessor muß das Sprungziel innerhalb eines Bereiches liegen, der durch die Grenzen

$$N - 128 \leqslant \text{Sprungziel} \leqslant N + 127$$

festgelegt ist. Die Bestimmung des Offset kann vom Anwender selbst durch Differenzbildung geschehen oder er überläßt dies einem Hilfsprogramm, das in Kapitel 6 als Assembler vorgestellt wird. Die Adressierungsart *relative* ist speicherplatzsparend, da nur 2 Worte nötig sind, um die Adresse *und* den Befehl eindeutig festzulegen.

Ist ein Sprung nur innerhalb der Grenzen einer Seite (page) möglich, so wird diese Adressierungsart als *page relative* bezeichnet.



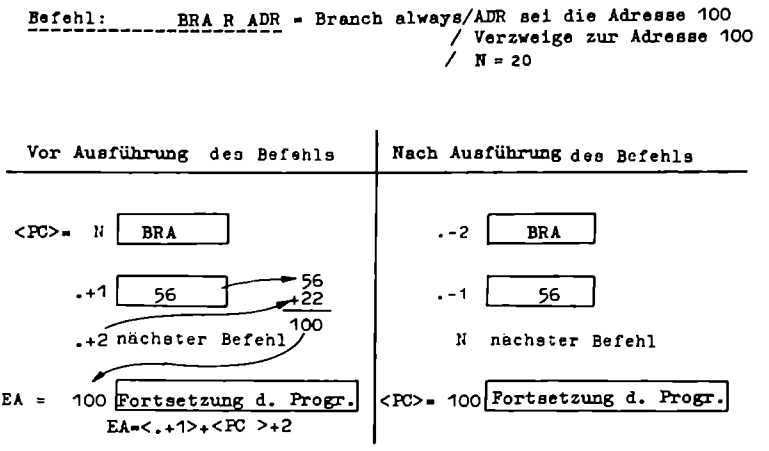
EA = ⟨. + 1⟩ + ⟨PC⟩ = ⟨. + 1⟩ + . + 2

(Beachte: dabei steht der PC auf ⟨PC⟩ + 2)

Symbol: R

Befehlsformat: 2

Beispiel zur Adressierungsart: Relative



Alle bisher behandelten Adressierungsarten sind mögliche Beispiele direkter Adressierung. Sie lassen sich unter dem Sammelbegriff „Direkte Adressierungsarten" zusammenfassen (Bild 3.5).

			EA des Operanden
REGISTERADRESSIERUNGSARTEN	1.	Register	Rn
	2.	Registerinhalt	< Rn >
	3.	Registerinhalt und Offset	< Rn > + < .+1 >
PROGRAMMZÄHLER-ADRESSIERUNGSARTEN	1.	Speicherplatz	.+1
	2.	Speicherwort	< .+1 >
	3.	Programmzählerinhalt und Speicherwort	<PC> + 2 + < .+1 >

Bild 3.5: Direkte Adressierungsarten (schematische Zusammenfassung)

3.6 Indirekte Adressierung

Definition

Unter *indirekter* Adressierung wird verstanden, daß nach der Adressenberechnung eine Speicheradresse verfügbar ist, welche die effektive Adresse des Operanden als Zeigeradresse enthält. Unter dieser Zeigeradresse (pointer address) ist der Operand zu finden. Die Zeigeradresse stellt die effektive Adresse des Operanden dar.

Die im folgenden behandelten Adressierungsarten lassen sich unter Anwendung obiger Definition aus den entsprechenden direkten Adressierungsarten ableiten. Für die Darstellung der effektiven Adresse (EA) der jeweiligen indirekten Adressierungsart wird die Verwendung der Zeigeradresse durch eine zusätzliche Klammerebene (( )) verdeutlicht.

Der indirekten Adressierung kommt für die Programmerstellung große Bedeutung zu, da die Adresse des Operanden eine Funktion der Zeigeradresse ist. Eine Änderung der Zeigeradresse bedingt, daß ein anderer Operand verarbeitet wird (siehe Abschnitt 4.12, Beispiele zu den Adressierungsarten). Befehle, die indirekte Adressierung von Operanden möglich machen, benötigen mehr Verarbeitungszeit, da der Speicher mehrmals adressiert werden muß, gestatten aber andererseits die Erstellung flexibler Programme (siehe Abschnitt 4.10, Argumententransfer).

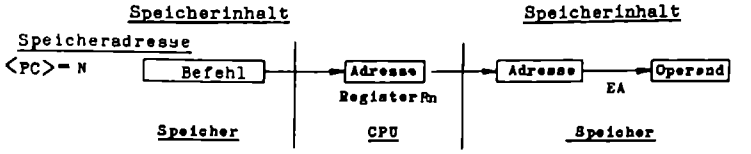
Ergänzend zur Definition der indirekten Adressierung noch zwei wichtige Anmerkungen:

1. Die Zeigeradresse kann ihrerseits noch für eine Adressenberechnung herangezogen werden (siehe 3.7.4 Adressierungsart *postindexed indirect*).
2. Der Umweg über die Zeigeradresse kann mehrmals wiederholt werden. Auf diese Weise ergibt sich eine mehrfach indirekte (multilevel indirect) Adressierung.

   Beispiel: Zweifach indirekte Adressierung. Die Zeigeradresse des Operanden (EA!) ist verfügbar, nachdem zweimal auf den Speicher zugegriffen wurde.

3.7 Indirekte Registeradressierungsarten

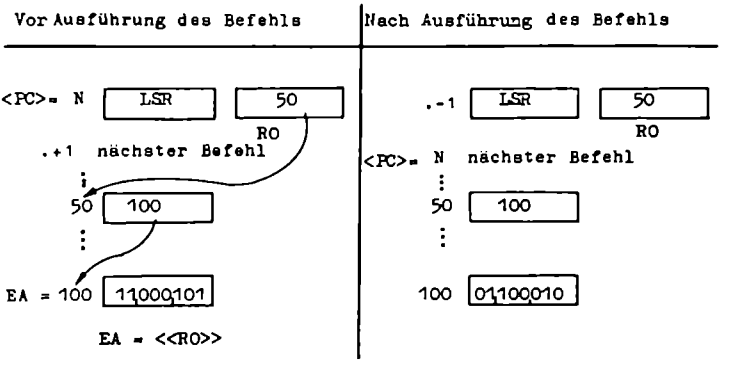
3.7.1 Register indirect



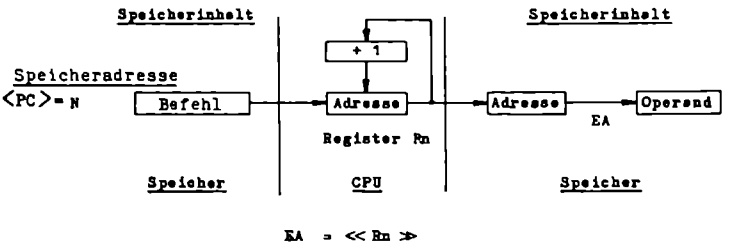
EA = ⟨⟨ Rn ⟩⟩
Symbol: I Rn
Befehlsformat: 3a (Spezialfall)

Beispiel zur Adressierungsart: Register indirect

```
Befehl:        LSR I R0 = Logical shift right
                          /Inhalt von << R0 >>um eine Stelle nach
                         /rechts schieben
```



3.7.2 Indirect autoincrement



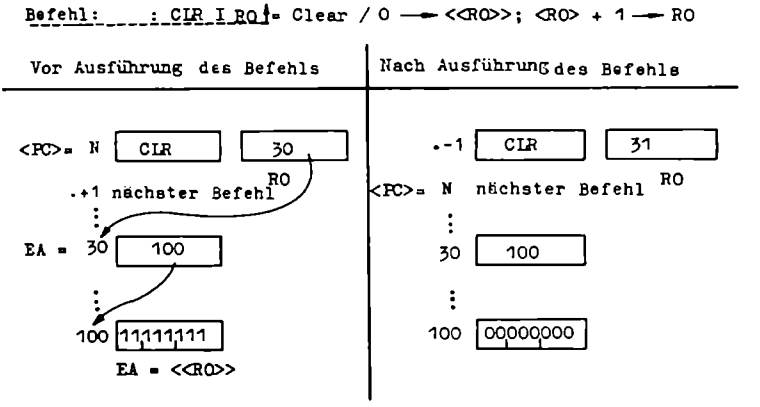
EA = ⟪ Rn ⟫

Nach Ausführung des Befehls: ⟨ Rn ⟩ + 1 → Rn

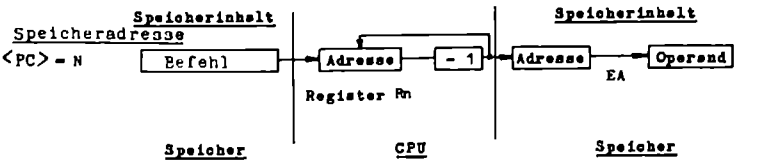
Symbol: I Rn ↑

Befehlsformat: 3a (Spezialfall)

Beispiel zur Adressierungsart: Indirect autoincrement



3.7.3 Indirect autodecrement



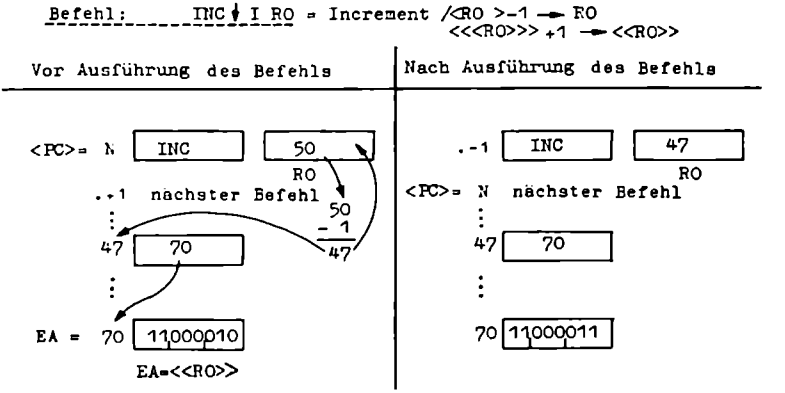
EA = ⟪Rn⟫

Vor Ausführung des Befehls : ⟨Rn⟩ - 1 → Rn

Symbol : ↓I Rn

Befehlsformat : 3a (Spezialfall)

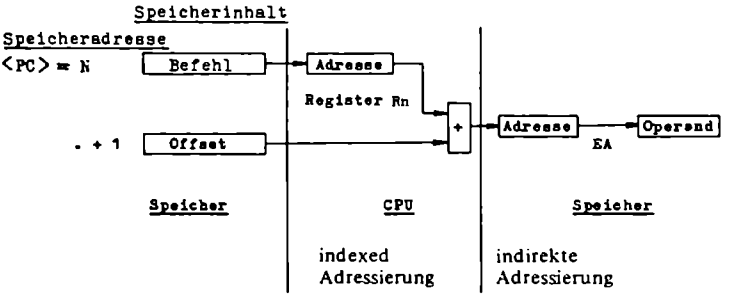
Beispiel zur Adressierungsart: Indirect autodecrement



3.7.4 Register indirect indexed

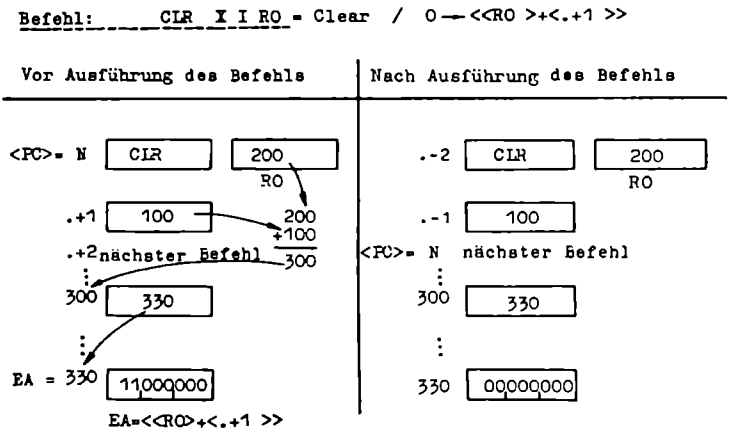
Bei dieser Adressierungsart unterscheidet man zwei Methoden, je nachdem, ob die *indexed* Adressierung vor (pre-) oder nach (post-) der indirekten bei der Bildung der effektiven Adresse durchgeführt wird.

Preindexed indirect Adressierung

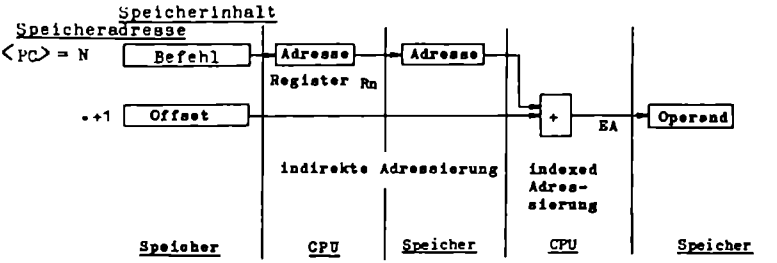


EA = ⟪ Rn ⟩ + ⟨ . + 1 ⟫
Symbol: X I Rn
Befehlsformat: 3a

Beispiel zur Adressierungsart: Preindexed indirect

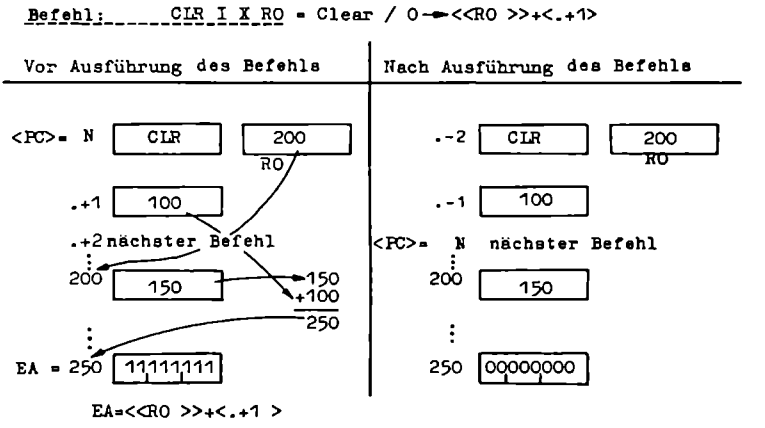


Postindexed indirect Adressierung



EA = ⟪Rn⟫ + ⟨.+1⟩
Symbol: I X Rn
Befehlsformat: 3a

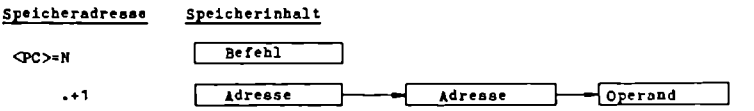
Beispiel zur Adressierungsart: Postindexed indirect



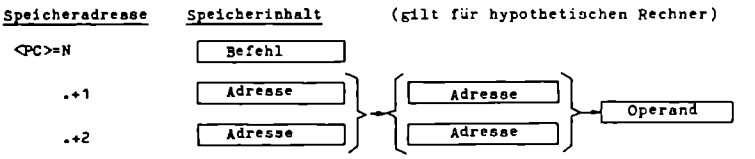
3.8 Indirekte Programmzähleradressierungsarten

3.8.1 Indirect

Diese Adressierungsart wird immer dort benötigt, wo die zur Verarbeitung anstehenden Dateien über eine *Liste von Adressen* zu erreichen sind. Sie leistet, daß die im Speicher zunächst gefundene Adresse zu einem Speicherplatz weist, an dem die Adresse des Operanden abgelegt ist. Damit der gesamte Speicherraum adressiert werden kann, müssen 2 Worte im Speicher vorgesehen sein, und zwar jeweils für die indirekte Adresse wie auch für die Zeigeradresse. Aus der unteren Anordnung wird klar, daß diese Adressierungsart bei der Ausführung viel Zeit benötigt, da der Speicher mehrmals adressiert werden muß (insgesamt sechsmal).

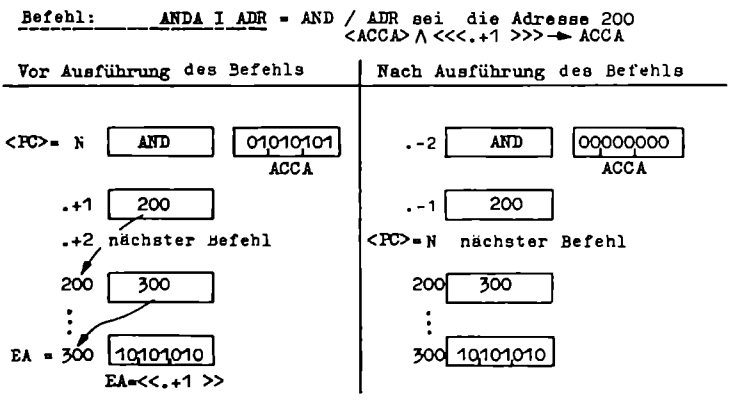


EA = ⟪.+1⟫
Symbol: I
Befehlsformat: 2



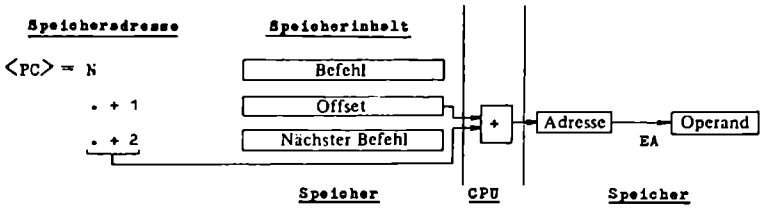
EA = ⟪.+1⟩⟨.+2⟫⟪.+1⟩⟨.+2⟩+1⟩
Symbol: I
Befehlsformat: 2

Beispiel zur Adressierungsart: Indirect



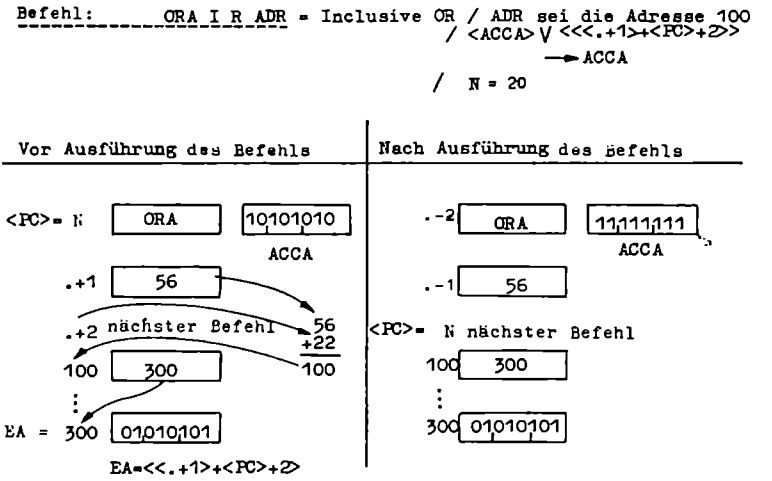
3.8.2 Indirect relative

Der Inhalt der Adresse, der sich aus der Summe von Offset und Programmzählerinhalt ergibt, ist die Zeigeradresse zum Operanden.



EA = ⟨(. + 1) + ⟨PC⟩ + 2⟩
Symbol: I R
Befehlsformat: 2

Beispiel zur Adressierungsart: Indirect relative



3.9 Häufige Mikroprozessor-Adressierungsarten

Es wurde im Verlauf dieses Kapitels immer wieder betont, wie wichtig gerade das Vorhandensein verschiedener Adressierungsmöglichkeiten bei einem Mikroprozessor ist.

Als unbedingt notwendige Adressierungsart wäre die *absolute* Adressierung zu bezeichnen, die auch bei jedem Mikroprozessor anzutreffen ist, weil mit ihr der Transfer zwischen Speicher und Zentraleinheit ermöglicht wird. Diese Adressierung würde an sich genügen, nur wäre ein solcher Mikroprozessor in seiner Leistungsfähigkeit entscheidend eingeengt, weil häufig Operanden adressiert werden müssen, die strukturiert angeordnet sind, oder weil Operanden schnell in den Mikroprozessor geladen werden müssen. Als weitere häufige Adressierungsarten sind die *indexed*, *Register direct*, *Register*, *autoincrement*, *autodecrement*, *relative*, *immediate*, *implied* und bei neueren Mikroprozessoren auch die *indirekte* Adressierung anzutreffen. Überhaupt ist festzustellen, daß der Bereitstellung vielseitiger Adressierungsarten von den Mikroprozessorherstellern immer mehr Bedeutung beigemessen wird. Dies hat natürlich zur Folge, daß die Architektur komplexer wird, da besonders die Anzahl der internen Register (Indexregister, Akkumulator) erhöht werden muß, um die benötigten Basisregister zur Verfügung zu stellen. Jedoch werden dadurch die Leistungsfähigkeit und die Einsatzmöglichkeiten für den Mikroprozessor erweitert (s. Kapitel 4).

Kapitel 4
Befehlsvorrat eines hypothetischen Mikroprozessors

Als Beispiel für einen Befehlsvorrat wird der eines hypothetischen 8-Bit-Mikroprozessors vorgestellt. Anhand dieses Befehlvorrats sei ein Überblick über die häufig in Mikroprozessoren anzutreffenden Befehle gegeben. Diese Aufstellung erhebt nicht den Anspruch der Vollständigkeit, denn es werden sich bei anderen Mikroprozessoren Befehle finden lassen, die von den angegebenen abweichen. Auch werden solche anzutreffen sein, mit denen sich die Programmierung spezieller Probleme leichter lösen läßt als mit denen des hypothetischen Mikroprozessors. In diesem Zusammenhang sei noch auf folgendes hingewiesen: Alle Befehle sind in ihrer englischen Bezeichnung angegeben, um so dem Leser eine möglichst schnelle Einarbeitung in die weiterführende Literatur zu ermöglichen.

4.1 Einteilung des Befehlsvorrats

Es besteht die Möglichkeit, den Befehlsvorrat nach verschiedenen Kriterien in Gruppen einzuteilen. Ein solches wäre z.B.: Welcher Teil des Mikroprozessorsystems wird von den betreffenden Befehlen angesprochen? Als Antwort auf diese Frage ergeben sich drei Gruppen:

1. Interne CPU-Einheiten,
2. Hauptspeicher,
3. Peripherie.

Man spricht in diesem Zusammenhang von CPU-bezogenen-, speicherbezogenen- und Eingabe/Ausgabe-Befehlen.

Um zu einer noch genaueren Auskunft über die Funktion eines Befehls zu gelangen, bedient man sich einer feineren Unterteilung.

Eine mögliche Unterteilung sei angegeben:

1. Arithmetische und logische Befehle
2. Vergleichs- und Testbefehle
3. Bitkontrollbefehle
4. Transferbefehle
5. Verschiebebefehle

6. Registerbefehle
7. Verzweigungs- und Sprungbefehle
8. Eingabe/Ausgabebefehle
9. Befehle zur Behandlung von Unterroutinen
10. Sonstige (Kontrollbefehle für Interrupt, Haltbefehl; NOP-Befehl (no operation))

Für das Verständnis des Befehlsvorrats ist die Kenntnis der internen Register des Mikroprozessors notwendig. Diese sind in Bild 4.1 dargestellt. Man erkennt das 8-Bit-Statusregister, zwei 8-Bit-Akkumulatoren A und B, ein 16-Bit-Register RO für Adressierungszwecke und drei 16-Bit-Register (Programmzähler, Indexregister, Stack Pointer).

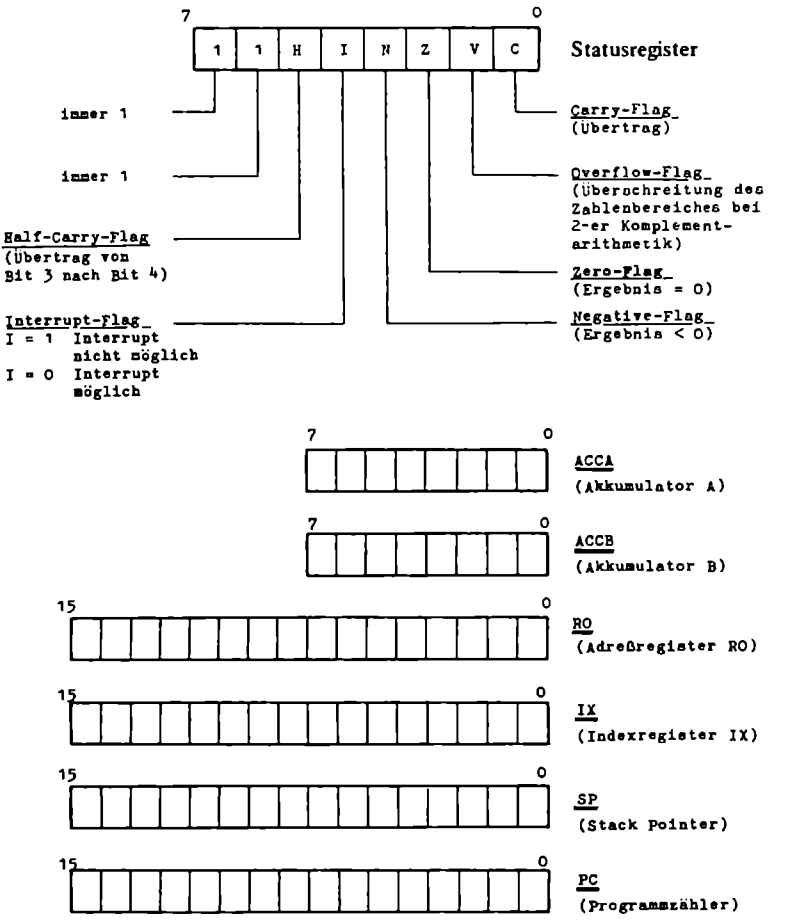


Bild 4.1: Interne Register des hypothetischen Mikroprozessors

Aus dieser Aufzählung verfügbarer interner Register werden das 8-Bit-Statusregister, das Adreßregister RO und der Stack Pointer näher erläutert.

Das Statusregister

Dieses Register kann als eine Summe von separaten Flip-Flops aufgefaßt werden, von denen jedes einzelne per Programm abgefragt und gesetzt werden kann. Die Bits des Statusregisters werden auch als Flags bezeichnet. Das Statusregister wird verwendet, um das Ergebnis einer Mikroprozessoroperation zu charakterisieren und um externe Operationen wie eine Programmunterbrechung (Interrupt) zu ermöglichen oder zu verhindern.

Wie aus den einzelnen Tabellen für die verschiedenen Befehle hervorgeht, werden die einzelnen Flags nicht nur von arithmetischen Operationen, sondern z.B. auch von Transferbefehlen (Tabelle 4.5) beeinflußt. Jedes Flag hat seine spezielle Bedeutung. Neben der Charakterisierung eines Ergebnisses werden die Flags noch von den Verzweigungs- und Sprungbefehlen benötigt, wann immer ein bedingter Sprung ausgeführt werden soll. Als bedingt wird ein Sprung oder eine Verzweigung dann bezeichnet, wenn sie abhängig von einer im Befehl gestellten Bedingung ausgeführt wird.

Im weiteren werden die einzelnen im Statusregister zusammengefaßten Flags besprochen.

Das Carry-Flag (C)

Das Carry-Flag zeigt einen Übertrag aus dem most significant bit (MSB) bei arithmetischen Operationen an. Wie später noch gezeigt wird, kommt dem Carry-Flag noch eine weitere Bedeutung bei den Verschiebebefehlen zu (Tabelle 4.6).

Das Zero-Flag (Z)

Dieses Flag zeigt erstens an, daß sich in einem der Akkumulatoren A oder B die Zahl Null befindet ($\langle ACCX \rangle = 0 \Rightarrow Z = 1$).

Ob die Zahl Null das Ergebnis einer arithmetischen Operation ist oder ob es sich um einen Transferbefehl handelt, spielt keine Rolle. Wichtig ist allein der Inhalt eines der Akkumulatoren.

Besonders zu beachten ist ein Zweites:

Das Zero-Flag wird weiterhin von Operationen beeinflußt, die sich auf das Indexregister beziehen (Tabelle 4.7). Damit ist eine Möglichkeit geschaffen, mit dem Indexregister bestimmte Programmtechniken zu verwirklichen (Schleifenzähler).

Das Zero-Flag kann im Gegensatz zum Carry-Flag nicht für sich allein, unabhängig von den anderen Flags, gesetzt werden. Dies ist jedoch keine wesentliche Einschränkung, da das Zero-Flag sowieso nur im Anschluß z.B. an eine arithmetische Operation abgefragt wird.

Das Overflow-Flag (V)

Dieses Flag ist wichtig, wenn in Zweierkomplementarithmetik gerechnet wird, d.h. mit Dualzahlen mit einem Vorzeichenbit (signed binary numbers). Bekanntermaßen ist bei einer 8-Bit-Dualzahl in Zweierkomplementdarstellung, wobei das MSB als Vorzeichenbit betrachtet wird, ein Zahlenbereich Z von $-128 \leqslant Z \leqslant +127$ darstellbar. Bei Arithmetikoperationen kann es nun ohne weiteres vorkommen, daß dieser

Zahlenbereich überschritten wird. Dies wird vom Overflow-Flag erkannt und für eventuelle Abfragen festgehalten. Die Bestimmung eines arithmetischen Overflow erfolgt für die *Addition* nach der folgenden Formel:

$$V_a = \overline{S_7} \wedge X_7 \wedge Y_7 \vee S_7 \wedge \overline{X_7} \wedge \overline{Y_7}$$
$$S = X + Y$$

Für die *Subtraktion* ergibt sich analog mit Y_7 ersetzt durch $\overline{Y_7}$:

$$V_s = \overline{D_7} \wedge X_7 \wedge \overline{Y_7} \vee D_7 \wedge \overline{X_7} \wedge Y_7$$
$$D = X - Y$$

Der Programmierer hat nun in einer Korrekturroutine dafür zu sorgen, daß die durch das Overflow-Flag angezeigte Zahlenbereichsüberschreitung berichtigt wird.

Das Negative-Flag (N)

Dieses Flag wird gesetzt abhängig vom most significant bit eines Ergebnisses (MSB = 1 N = 1; MSB = 0; N = 0).

Seine eigentliche Bedeutung erhält das Negative-Flag, sobald Zweierkomplementarithmetik durchgeführt wird.

Das MSB repräsentiert das Vorzeichen einer Dualzahl in Zweierkomplementdarstellung. Ein gesetztes MSB (MSB = 1) zeigt an, daß die Dualzahl negativ ist. Eine positive Dualzahl liegt vor, wenn das MSB = 0 ist (Tabelle 4.6).

Zur Feststellung eines Overflows bei Multiplikation und Division vorzeichenbehafteter Dualzahlen wird auf das N-Flag zurückgegriffen (siehe Abschnitt 4.6). Ein Anwendungsbereich, in dem das Negative-Flag, in Verbindung mit dem Carry-Flag, eine wichtige Rolle spielt, sind die Verschiebebefehle.

Das Negative-Flag darf aber keineswegs nur in Verbindung mit arithmetischen Operationen gesehen werden. Es hat außerdem noch die Aufgabe, den Zustand des MSB anzuzeigen, unabhängig von der vorausgegangenen Operation.

Das Half-Carry-Flag (H)

Dieses Flag wird abhängig von den Additionsbefehlen gesetzt, und zwar, wenn ein Übertrag von Bit 3 nach Bit 4 erfolgte. Werden diese Additionsbefehle im Zusammenhang mit BCD-Zahlen verwendet, so wird unter anderem das Half-Carry-Flag herangezogen, um mittels des DAA-Befehls das Ergebnis notfalls ins BCD-Format überzuführen.

Das Interrupt-Mask-Flag (I)

Dieses Flag kann per Programm gesetzt und gelöscht werden. Seine Aufgabe besteht darin, eine Programmunterbrechung (Interrupt) zu verhindern (I = 1) oder zuzulassen (I = 0). Der Interrupt wird in Kapitel 5 noch genauer untersucht. Mit dem Interrupt-Mask-Flag ist eine Steuerung des Interruptsystems des Mikroprozessors möglich.

Zusammenfassend läßt sich sagen, daß mit dem Statusregister und den damit verbundenen Befehlen dem Programmierer eine Möglichkeit zur Hand gegeben ist, mit der der Programmablauf überwacht und gesteuert werden kann. Man denke hierbei an die noch zu behandelnden bedingten Verzweigungs- und Sprungbefehle (Tabelle 4.8).

Neben den vorgestellten Flags gibt es noch weitere, wie z.B. Parity-Flag, Decimal-Mode-Flag.

Die vorgestellten Flags beziehen sich allesamt auf den internen Status des Mikroprozessors. Sie haben die Aufgabe, Ergebnisse durchgeführter Operationen zu charakterisieren.

Das Adreßregister RO

Das Adreßregister RO, welches 16 Bit breit ist, dient als Basisregister für die im hypothetischen 8-Bit-Mikroprozessor definierten Registeradressierungsarten. Unter Verwendung von RO kann der gesamte Speicherraum adressiert werden. Zur Adressierungsart *indirect autoincrement* bzw. *autodecrement* sie folgendes bemerkt:

Die EA des Operanden ist als «RO» angegeben, entsprechend dem Symbol in Kapitel 3, Bild 3.4. Da es sich hier jedoch um einen 8-Bit-Mikroprozessor handelt, muß die Zeigeradresse in zwei aufeinanderfolgenden Speicherplätzen abgelegt sein, um den gesamten Speicherbereich von 64 k indirekt adressieren zu können.

Der Stack Pointer

Der Stack Pointer stellt ebenso wie der Programmzähler ein Register dar, dessen Breite mit der Anzahl der Adreßleitungen identisch ist. Dieses Register überwacht einen definierten Speicherbereich, den Stack (Stapel engl.: Stack), der sich außerhalb des Mikroprozessors in einem Schreib-Lese-Speicher (random access memory (RAM)) befindet. Ein anderer Name für diese Art von Stack ist *Software- oder Pointer-Stack.*

Diese Stackarchitektur wurde für den hypothetischen Mikroprozessor gewählt. Eine andere Möglichkeit, den Stack zu realisieren, ist der *Hardware-Stack.* Dieser befindet sich als Registerbereich innerhalb des Mikroprozessors. Ein Nachteil dieser Stackarchitektur ist in der begrenzten Anzahl interner Register zu sehen. Der Hardware-Stack bietet jedoch den Vorteil schneller Transfers der Operanden vom Stack zur ALU, da die internen Register, die den Stack bilden, mit der ALU verbunden sind und so ein Speicherzyklus vermieden wird. Weiterhin wird für einen Programmabschnitt oft nur ein kleiner Teil des Stacks benötigt, so daß die Operanden in den internen Registern abgelegt werden können.

Sowohl dem Hardware- als auch dem Software-Stack ist folgendes Prinzip eigen: last in first out (LIFO).

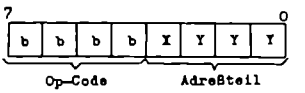
Dieses Prinzip besagt, der zuletzt auf dem Stack abgespeicherte Operand wird als erster wieder ausgelesen. Oft wird der Stack auch als „push-down list" bezeichnet.

Der Stack wird innerhalb eines Mikrocomputers zu vielen Aufgaben herangezogen:

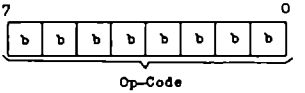
- Abspeichern von Daten (z.B. Zwischenergebnisse).
- Abspeichern von Adressen (z.B. Rückkehradressen bei Unterprogrammanrufen, Adressen von Operanden).
- Abspeichern des internen Status der CPU (z.B. bei Programmunterbrechung (Interrupt) von externen Geräten, Spannungsausfall).
- Ermöglicht automatisches subroutine nesting (= eine Unterroutine ruft eine andere Unterroutine auf).
- Erleichtert das Schreiben von reentrant subroutines (= Unterprogramme, die fehlerlos arbeiten, wenn sie von einem Interrupt unterbrochen werden, dessen Bedienungsprogramm dieselbe Unterroutine anruft).

1. Zweiadreßbefehle

1a) Mit mehreren Adressierungsarten

b b b b X Y Y Y Op-Code AdreßTeil	Mnemonic	Befehlsformat	Adressierungs-art
0 0 0 0 X 0 Y Y	ADDX		Register Immediate Extended Indirect Indexed Autoincrement Autodecrement Indirect Autoincrement Indirect Autodecrement
0 0 0 1 X 0 Y Y	ADCX		
0 0 1 0 X 0 Y Y	SUBX		
0 0 1 1 X 0 Y Y	SBCX		
0 1 0 0 X 0 Y Y	ANDX		
0 1 0 1 X 0 Y Y	EORX		
0 1 1 0 X 0 Y Y	ORAX		
0 1 1 1 X 0 Y Y	BITX		
1 0 0 0 X 0 Y Y	CMPX		
1 0 0 1 X Y Y Y	LDAX		
1 0 1 0 X Y Y Y	STAX		

1b) Mit einer Adressierungsart

b b b b b b b b Op-Code	Mnemonic	Befehlsformat	Adressierungs-art
0 0 0 0 0 1 1 1	ABA		Implied
0 0 1 1 0 1 1 1	SBA		
1 0 0 0 0 1 0 0	CBA		
1 0 0 0 0 1 1 0	TBA		
1 0 0 0 1 1 1 0	TAB		
1 0 0 0 0 1 1 1	TPA		
1 0 0 0 0 1 0 1	TAP		
0 0 1 1 0 1 0 0	TSX		
0 0 1 1 1 1 0 0	TXS		

2. Einadreßbefehle

2a) Mit mehreren Adressierungsarten

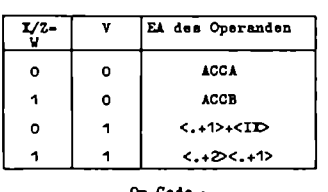
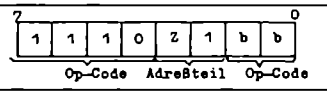
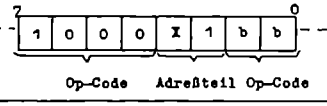
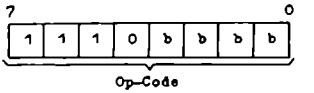
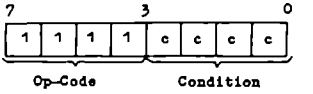
b b b b W V b b AdreßTeil Op-Code	Mnemonic	Befehlsformat	Adressierungs-art
1 0 1 1 W V 0 0	COM		Register Indexed Extended
1 0 1 1 W V 0 1	NEG		
1 0 1 1 W V 1 0	CLR		
1 0 1 1 W V 1 1	TST		
1 1 0 0 W V 0 0	DEC		
1 1 0 0 W V 0 1	INC		
1 1 0 0 W V 1 0	ROL		
1 1 0 0 W V 1 1	ROR		
1 1 0 1 W V 0 0	ASL		
1 1 0 1 W V 0 1	ASR		
1 1 0 1 W V 1 0	LSR		
1 1 0 1 0 0 1 1	DAA		
1 1 1 0 Z 1 0 0	LDX		Indexed Extended
1 1 1 0 Z 1 0 1	LDS		
1 1 1 0 Z 1 1 1	LDR		
1 0 0 0 X 1 0 1	PSHX		Autodecrement Autoincrement
1 0 0 0 X 1 0 0	PULX		

Bild 4.2a: Befehlsformate des hypothetischen Mikroprozessors

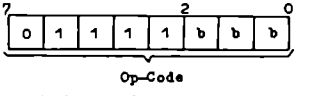
2b) Mit einer Adressierungsart

Op-Code	Mnemonic	Befehlsformat	Adressierungsart
1 1 1 0 0 0 0	STX		Implied
1 1 1 0 0 0 1	STS		
1 1 1 0 1 1 0	STR		
1 1 1 0 0 1 1	CPX		
1 1 1 0 0 1 0	DEX		
1 1 1 0 0 1 0	INX		
1 1 1 0 1 0 0	DES		
1 1 1 0 1 0 1	INS		
1 1 1 0 1 1 0	DER		
1 1 1 0 1 1 1	INR		

3. Verzweigungsbefehle

Op-Code	Condition	Mnemonic	Befehlsformat	Adressierungsart
1 1 1 1	0 0 0 1	BRA		Relative
1 1 1 1	0 0 1 0	BHT		
1 1 1 1	0 0 1 1	BLS		
1 1 1 1	0 1 0 0	BCC		
1 1 1 1	0 1 0 1	BCS		
1 1 1 1	0 1 1 0	BNE		
1 1 1 1	0 1 1 1	BEQ		
1 1 1 1	1 0 0 0	BVC		
1 1 1 1	1 0 0 1	BVS		
1 1 1 1	1 0 1 0	BPL		
1 1 1 1	1 0 1 1	BMI		
1 1 1 1	1 1 0 0	BGE		
1 1 1 1	1 1 0 1	BLT		
1 1 1 1	1 1 1 0	BGT		
1 1 1 1	1 1 1 1	BLE		

4. Bitkontrollbefehle

Op-Code	Mnemonic	Befehlsformat	Adressierungsart
0 1 1 1 1 0 1 0	CLV		Implied
0 1 1 1 1 0 1 1	SEV		
0 1 1 1 1 1 0 0	CLC		
0 1 1 1 1 1 0 1	SEC		
0 1 1 1 1 1 1 0	CLI		
0 1 1 1 1 1 1 1	SEI		

5. Formate sonstiger Befehle

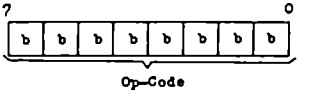
Op-Code	Mnemonic	Befehlsformat	Adressierungsart
0 1 0 1 z 1 0 0	JSR		Indexed Extended
0 1 0 0 z 1 0 0	JMP		
0 1 0 1 1 1 0 1	RTS		Implied
0 1 0 1 1 1 1 0	BSR		Relative
0 1 1 0 1 1 1 1	RTI		Implied
1 1 1 1 0 0 0 0	NOP		Implied

Bild 4.2b: Befehlsformate des hypothetischen Mikroprozessors

EA = effective address

X	Adressierungsart	Akkumulator
0	Register bzw.	A
1	Implied	B

YYY	Adressierungsart	EA des Operanden	Symbol
000	Immediate	.+1	#
001	Extended	⟨.+2⟩⟨.+1⟩	E
010	Indirect	⟨⟨.+1⟩⟨.+2⟩⟩ ⟨⟨.+1⟩⟨.+2⟩+1⟩	I
011	Indexed	⟨.+1⟩ + ⟨IX⟩	X
100	Autoincrement	⟨R0⟩	R0↑
101	Autodecrement	⟨R0⟩	↓R0
110	Indirect Autoincrement	⟨⟨R0⟩⟩	IR0↑
111	Indirect Autodecrement	⟨⟨R0⟩⟩	↓IR0

Z	Adressierungsart	EA des Operanden	Symbol
0	Indexed	⟨.+1⟩ + ⟨IX⟩	X
1	Extended	⟨.+2⟩⟨.+1⟩	E

Bild 4.2c: Adressierungsarten des hypothetischen Mikroprozessors

Der Befehlsvorrat wird in die schon genannten Gruppen eingeteilt. Jeder Befehl ist gekennzeichnet durch eine ihm eigene Abkürzung, das sogenannte *Mnemonic*. Dieses stellt eine Kurzschreibweise für den jeweiligen Befehl dar und trägt wesentlich zum leichteren Verständnis eines Programms bei. Aus Bild 4.2 a, b ist ein wichtiger Zusammenhang zwischen Mnemonic, Adressierungsart und Binärformat zu erkennen:

Zu einem Befehl, gekennzeichnet durch Mnemonic und Adressierungsart, gehört genau ein Bitmuster (binäres Format), z.B. bestehend aus Op-Code und Adreßteil. Von dieser Relation macht in Kapitel 6 der Assembler, ein Hilfsprogramm zur Softwareerstellung, ausgiebig Gebrauch. Die Tabellen in Bild 4.2 a, b geben zu jedem Befehl das Befehlsformat an. Dieses läßt erkennen, wie für die einzelnen Befehle die zugehörigen Bitmuster abhängig von den jeweils definierten Adressierungsarten gebildet werden. In Bild 4.2 c sind die Adressierungsarten des hypothetischen Mikroprozessors angegeben, deren Funktion in Kapitel 3 erklärt wurde.

Die Tabellen für die einzelnen Befehle enthalten neben dem Mnemonic und der Funktionsbeschreibung auch Angaben darüber, wie das Statusregister von den betreffenden Befehlen beeinflußt wird. Zur Funktionsbeschreibung wurde die Klammerschreibweise aus Kapitel 2 verwendet.

Die Zeichen unter der Rubrik „Statusregister" drücken aus, welches Kontrollbit bei einer Operation beeinflußt wird. Dabei bedeutet:

0 = Löschen
1 = Setzen
– = keine Beeinflussung
↕ = wird abhängig von der Operation entweder gelöscht oder gesetzt.

4.2 Arithmetische Befehle

Tabelle 4.1 enthält vornehmlich Befehle, die sich auf die beiden Akkumulatoren A und B beziehen. Beide Akkumulatoren werden miteinander oder mit einer durch M gekennzeichneten Speicherzelle über arithmetische Befehle verknüpft. Die Ergebnisse werden mit Hilfe der Kontrollbits des Statusregisters gekennzeichnet, und zwar so, daß angegeben wird, ob ein Übertrag (C), ein arithmetischer Overflow (V), ein Half-Carry (H) aufgetreten sind oder ob das Ergebnis Null (Z) beziehungsweise negativ (N) ist. Aus Tabelle 4.1 ist zu entnehmen, wann immer ein Bit des Statusregisters beeinflußt wird. Wichtig ist hierbei, daß das Statusregister unmittelbar nach einer arithmetischen Operation gesetzt wird. Für eventuelle bedingte Verzweigungen hat dies zur Folge, daß die Abfrage: Verzweigen? Ja oder Nein, spätestens vor dem nächsten Befehl erfolgen muß, der das Statusregister verändert. Sonst müssen besondere Maßnahmen ergriffen werden, um das Ergebnis zu speichern, damit es zu einem späteren Zeitpunkt auf bestimmte Eigenschaften (Null, negativ) überprüft werden kann.

Einige Befehle, die diese Prüfung leisten, werden im Abschnitt über Vergleichs- und Testbefehle vorgestellt.

Zuletzt sei noch der DAA-Befehl, der letzte in Tabelle 4.1, erläutert. Er wird verwendet, um bei der Addition zweier BCD-Zahlen das Ergebnis, sofern nötig, ins BCD-Format umzuwandeln, d.h. zu korrigieren. Denn eine Besonderheit des BCD-Code besteht darin, daß nur die 4-Bit-Dualzahlen $0000_2 = 0_{10}$ bis $1001_2 = 9_{10}$ definiert sind, während die Dualzahlen 1010_2 bis 1111_2 nicht definiert sind. Aus diesem Grunde muß nach einer Addition zweier BCD-Zahlen, die in binär durchgeführt wird, dafür Sorge getragen werden, daß das Ergebnis wieder ins BCD-Format übergeführt wird. Der DAA-Befehl bedient sich dabei zweier Tests. Erstens wird geprüft, ob eine 4-Bit-Dualzahl größer als 1001_2 ist, zweitens, ob ein Übertrag aus dem Half-Carry-Bit oder dem Carry-Bit existiert. Das Half-Carry-Bit zeigt einen Übertrag aus dem 3. Bit an, während das Carry-Bit den aus dem 7. Bit anzeigt.

Fällt einer dieser Tests positiv aus, so wird zur oberen bzw. unteren Hälfte der 8-Bit-Dualzahl $0110_2 = 6_{10}$ hinzuaddiert (Korrekturfaktor).

Die Funktion des DAA-Befehls kann wie folgt zusammengefaßt werden:

DAA: Addition von 0110_2, wenn
$b_3 b_2 b_1 b_0 > 1001_2 \vee H = 1$ ist;
Addition von $0110\ 0000_2$, wenn
$b_7 b_6 b_5 b_4 > 1001_2 \vee C = 1$ ist.
Sind obige Bedingungen nicht erfüllt, so entfällt diese Addition.

DAA steht für Decimal adjust accumulator.

Tabelle 4.1: Arithmetische Befehle

Mnemonic	Funktionsbeschreibung		Statusregister H	I	N	Z	V	C
ADDA	Add	(ACCA) + (M) → ACCA	↕	–	↕	↕	↕	↕
ADDB		(ACCB) + (M) → ACCB	↕	–	↕	↕	↕	↕
ABA	Add ACCB to ACCA	(ACCA) + (ACCB) → ACCA	↕	–	↕	↕	↕	↕
ADCA	Add with carry	(ACCA) + (M) + (C)→ACCA	↕	–	↕	↕	↕	↕
ADCB		(ACCB) + (M) + (C)→ACCB	↕	–	↕	↕	↕	↕
COM	Complement (one's)	$\overline{(M)}$ → M	–	–	↕	↕	0	1
COMA		$\overline{(ACCA)}$ → ACCA	–	–	↕	↕	0	1
COMB		$\overline{(ACCB)}$ → ACCB	–	–	↕	↕	0	1
NEG	Complement (two's)	–(M) → M	–	–	↕	↕	↕	↕
NEGA		–(ACCA) → ACCA	–	–	↕	↕	↕	↕
NEGB		–(ACCB) → ACCB	–	–	↕	↕	↕	↕
SUBA	Subtract	(ACCA) – (M) → ACCA	–	–	↕	↕	↕	↕
SUBB		(ACCB) – (M) → ACCB	–	–	↕	↕	↕	↕
SBA	Subtract ACCB from ACCA	(ACCA) – (ACCB) → ACCA	–	–	↕	↕	↕	↕
SBCA	Subtract with carry	(ACCA) – (M) – (C)→ACCA	–	–	↕	↕	↕	↕
SBCB		(ACCB) – (M) – (C)→ACCB	–	–	↕	↕	↕	↕
CLR	Clear	0 → M	–	–	0	1	0	0
CLRA		0 → ACCA	–	–	0	1	0	0
CLRB		0 → ACCB	–	–	0	1	0	0
DEC	Decrement	(M) – 1 → M	–	–	↕	↕	↕	–
DECA		(ACCA) – 1 → ACCA	–	–	↕	↕	↕	–
DECB		(ACCB) – 1 → ACCB	–	–	↕	↕	↕	–
INC	Increment	(M) + 1 → M	–	–	↕	↕	↕	↕
INCA		(ACCA) + 1 → ACCA	–	–	↕	↕	↕	–
INCB		(ACCB) + 1 → ACCB	–	–	↕	↕	↕	–
DAA	Decimal Adjust Accumulator	Addition des Korrekturfaktors 0110: wenn $b_3b_2b_1b_0 > 1001_2$ V H = 1 oder $b_7b_6b_5b_4 > 1001_2$ V C = 1	–	–	↕	↕	↕	–

M = Speicherplatz

4.3 Logische Befehle

Diese in Tabelle 4.2 vorgestellten Befehle führen logische Operationen der Form

$$\langle ACCX\rangle \begin{bmatrix}\wedge\\ \veebar\\ \vee\end{bmatrix} \langle M\rangle \rightarrow ACCX$$

durch. Die logischen Operationen umfassen die Verknüpfungen: UND ($\wedge$), Exklusiv-ODER ($\veebar$) und Inklusiv-ODER ($\vee$).

Diese Verknüpfungen werden zwischen einem der beiden Akkumulatoren A und B und einem durch M bestimmten Speicherplatz bitweise durchgeführt. Die Rubrik „Statusregister" zeigt, daß nur das Negative- und das Zero-Flag beeinflußt werden. Das Ergebnis einer boolschen Verknüpfung wird in ACCX abgespeichert, der dort stehende Operand wird überschrieben, „zerstört". Über die beiden Flags Negative und Zero wird angezeigt, ob das MSB = 1 und ob $\langle ACCX\rangle = 0$ war.

Der logische Befehl AND ($\wedge$) eignet sich zum Ausblenden bestimmter Bits eines Akkumulators, die danach anhand des Negative- und Zero-Flags abgefragt werden können. Die in M abgelegte Konstante wird als Maske bezeichnet.

Bei den logischen Befehlen Exclusive OR ($\veebar$) und Inclusive OR ($\vee$) werden die Bits eines Akkumulators entsprechend den boolschen Verknüpfungen gebildet. Exclusive OR wird häufig zum bitweisen Invertieren verwendet. Für jedes Bit, das invertiert werden soll, steht in der entsprechenden Stelle im Speicherplatz M eine binäre 1. Danach ist ebenfalls eine Abfrage der beeinflußten Flags des Statusregisters möglich.

Die logischen Befehle ermöglichen den gezielten Zugriff zu einzelnen Bits der Akkumulatoren A und B.

Tabelle 4.2: Logische Befehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
ANDA	AND	<ACCA > $\wedge$ <M > → ACCA	- - ↕ ↕ 0 -
ANDB		<ACCB > $\wedge$ <M > → ACCB	- - ↕ ↕ 0 -
EORA	Exclusive OR	<ACCA > $\veebar$ <M > → ACCA	- - ↕ ↕ 0 -
EORB		<ACCB > $\veebar$ <M > → ACCB	- - ↕ ↕ 0 -
ORA	Inclusive OR	<ACCA > $\vee$ <M > → ACCA	- - ↕ ↕ 0 -
ORB		<ACCB > $\vee$ <M > → ACCB	- - ↕ ↕ 0 -

4.4 Bitkontroll- und Testbefehle

Diese Befehle aus den Tabellen 4.3 und 4.4 verändern das Statusregister.

4.4.1 Bitkontrollbefehle

Die Befehle SEC (set carry) und CLC (clear carry) beeinflussen das Carry-Flag, es kann gesetzt und gelöscht werden.

Ebenso kann mit den Befehlen CLV (clear overflow) und SEV (set overflow) das Overflow-Flag verändert werden.

Eine Programmunterbrechung wird abhängig von den Befehlen CLI und SEI zugelassen oder verhindert.

Tabelle 4.3: Bitkontrollbefehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
CLC	Clear carry	0 → C	- - - - - 0
CLI	Clear interrupt mask	0 → I	- 0 - - - -
CLV	Clear overflow	0 → V	- - - - 0 -
SEC	Set carry	1 → C	- - - - - 1
SEI	Set interrupt mask	1 → I	- 1 - - - -
SEV	Set overflow	1 → V	- - - - 1 -
TAP	Transfer contents of ACCA to status register	<ACCA>V 11000000 → CCR	b_5 b_4 b_3 b_2 b_1 b_0
TPA	Transfer contents of status register to ACCA	<CCR > V 11000000 → ACCA	

Tabelle 4.4: Vergleichs- und Testbefehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
BITA	Bit Test	<ACCA > ∧ <M >	- - ↕ ↕ 0 -
BITB		<ACCB > ∧ <M >	- - ↕ ↕ 0 -
CMPA	Compare	<ACCA >-<M >	- - ↕ ↕ ↕ ↕
CMPB		<ACCB >-<M >	- - ↕ ↕ ↕ ↕
CBA	Compare ACCB with ACCA	<ACCA >-<ACCB >	- - ↕ ↕ ↕ ↕
TST	Test zero or minus	<M >-0	- - ↕ ↕ 0 0
TSTA		<ACCA >-0	- - ↕ ↕ 0 0
TSTB		<ACCB >-0	- - ↕ ↕ 0 0

Für den Programmierer sind besonders die Befehle TAP und TPA nützlich. TAP lädt den Akkumulatorinhalt in das Statusregister. Dadurch können dessen einzelne Bits zugleich verändert werden. Mit dem Befehl TPA wird das Statusregister in den Akkumulator A kopiert. Dort können mit Vergleichs- und Testbefehlen alle Bits des Statusregisters auf einmal abgefragt werden, d.h. es wird auf ein Bitmuster verglichen (Compare-Befehl in Tabelle 4.4).

4.4.2 Testbefehle

Die Befehle BITA und BITB beeinflussen das Negative- und Zero-Flag und setzen das Overflow-Flag zurück (Tabelle 4.4).

Diese Befehle führen das logische UND zwischen einem Operanden in ACCX und in M durch. Dabei bleiben beide Operanden erhalten. Diese Art von nicht „zerstörenden" Tests ist oft notwendig, um einen Operanden auf ein spezielles Bit hin zu testen. Da dieser Test bitweise erfolgt, kann eine Maske in einen der Akkumulatoren geladen werden, die dafür sorgt, daß nach dem logischen UND zwischen Akkumulator und Speicherplatzinhalt (⟨M⟩) nur die nicht maskierten Bits die Flags (N und Z) des Statusregisters beeinflussen. Damit kann auf sehr einfache Weise z.B. der Betriebszustand(Status) eines Peripheriegerätes abgefragt werden. Anschließend kann mit einem bedingten Sprungbefehl in ein Programm gesprungen werden, das das Peripheriegerät gemäß seinem Status bedient.

Die Befehle TST, TSTA, TSTB beeinflussen wiederum das Negative- und Zero-Flag. Hierbei wird jedoch nur geprüft, ob der Operand $\leqslant 0$ ist, d.h. es besteht nicht die Möglichkeit, über eine Maske bestimmte Bits zu testen. Diese Testbefehle sind für die Fälle vorgesehen, in denen nicht unmittelbar nach einer arithmetischen Operation auf Null beziehungsweise negativ geprüft werden kann, z.B. wenn die Akkumulatoren A und B geladen werden und zunächst mit A gearbeitet wird. Anschließend soll nun, abhängig davon, ob ⟨B⟩ = 0 oder < 0 war, eine Programmverzweigung vorgenommen werden. Dies erfolgt nun mit TSTB und einer anschließenden bedingten Verzweigung. Diese Befehle ermöglichen neben dem Test der Akkumulatoren auch den Test beliebiger Speicherplätze. Dies ist wichtig, wenn es sich, wie im vorliegenden Fall, um einen Mikroprozessor handelt, der Eingabe- und Ausgaberegister wie Speicherplätze behandelt. Somit ist es einfach möglich, den Status eines Peripheriegerätes abzufragen, da dieser häufig einer Speicherstelle zugeordnet ist.

Die Befehle CMPA und CMPB sowie CBA werden dort eingesetzt, wo zwei Operanden miteinander verglichen werden sollen. Die beiden Operanden werden voneinander subtrahiert, wobei der Status des Ergebnisses über das Zero-, Negative-, Carry- und Overflow-Flag angezeigt wird. Mit einem anschließenden bedingten Verzweigungsbefehl kann abhängig vom Ergebnis in den weiteren Programmverlauf eingegriffen werden.

4.5 Transferbefehle

Mit den in Tabelle 4.5 aufgeführten Befehlen wird ermöglicht, daß Operanden von den Akkumulatoren an einen durch die Adressierungsart spezifizierten Speicherplatz geschrieben werden oder in umgekehrter Weise von dort in die Akkumulatoren geladen werden. Des weiteren ist die Möglichkeit eines schnellen Austausches der Akkumulatorinhalte untereinander gegeben (TAB, TBA). Durch die Transferbefehle werden das Negative- und das Zero-Flag beeinflußt.

Tabelle 4.5: Transferbefehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
LDAA	Load Accumulator	<M> → ACCA	- - ↕ ↕ 0 -
LDAB		<M> → ACCB	- - ↕ ↕ 0 -
STAA	Store Accumulator	<ACCA> → M	- - ↕ ↕ 0 -
STAB		<ACCB> → M	- - ↕ ↕ 0 -
TAB	Transfer ACCA to ACCB	<ACCA> → ACCB	- - ↕ ↕ 0 -
TBA	Transfer ACCB to ACCA	<ACCB> → ACCA	- - ↕ ↕ 0 -

4.6 Verschiebebefehle

In Tabelle 4.6 sind fünf Gruppen verschiedener Verschiebebefehle beschrieben. Sie beziehen sich allesamt auf beide Akkumulatoren A und B und auf einen Speicherplatz M. Bei der Behandlung der 4-Bit-Akkumulator-Arithmetikeinheit in Kapitel 2 wurden diese Befehle bereits vorgestellt.

Ihre Anwendungsgebiete umfassen neben der Möglichkeit des Links- und Rechtsverschiebens auch arithmetische Operationen (Multiplikation, Division mit 2-er Potenzen). Die Tabelle 4.6 zeigt außerdem, daß alle Verschiebebefehle die Flags N, Z, V und C beeinflussen. Diese Befehle ermöglichen außerdem, einzelne Bits nach einer Verschiebung zu testen. Abhängig vom Zustand des betreffenden Bits kann alternativ verzweigt werden (Tabelle 4.8).

Die oben erwähnten fünf Gruppen aus Tabelle 4.6 werden im folgenden vorgestellt. Die Gruppen enthalten rotierende Verschiebebefehle (ROL, ROR), arithmetische Verschiebebefehle (ASL, ASR) und den logischen Verschiebebefehl (LSR).

Die rotierenden Verschiebebefehle

Die Befehle ROL und ROR sind dadurch gekennzeichnet, daß in die Links- und Rechtsverschiebung jeweils das Carry-Flag mit einbezogen ist. Bei ROL wird das Carry-Flag nach b_0 geschoben und b_7 wird ins Carry-Flag geladen. Bei ROR ver-

läuft diese Operation in umgekehrter Richtung ((C) → b_7 (b_0) → C). Aufgrund der bestehenden Rückkopplung werden diese Verschiebebefehle als „rotierende" bezeichnet (→ Name: Rotate). Dadurch wird es möglich, bestimmte Bits ins Carry-Flag oder das MSB zu verschieben, die anschließend über das Statusregister abgefragt werden (N, Z, V, C!).

Manchmal finden die Befehle ROL und ROR bei der Multiplikation (ROL) und der Division (ROR) von Dualzahlen ohne Verzeichenbit (unsigned binary numbers) Verwendung. ROL und ROR werden daher oft auch als *unsigned shifts* bezeichnet. Wird eine Multiplikation mit 2 durchgeführt, so muß das Carry-Flag vor der Ausführung des Befehls ROL zurückgesetzt werden, denn das Carry-Flag wird nach b_0 geschoben (Rotate!). Ein gesetztes Carry-Flag nach der Multiplikation deutet auf einen Overflow, d.h. eine Zahlenbereichsüberschreitung hin.

Die arithmetischen Verschiebebefehle

Bei den Befehlen ASL und ASR fällt auf, daß es keine Rückführung des Carry nach b_0 bzw. nach b_7 gibt.

ASL bietet sich an zur Durchführung einer Multiplikation mit 2, wenn Zweierkomplementarithmetik vorliegt, d.h. Dualzahlen mit Vorzeichenbit (signed binary numbers).

ASR führt eine Division einer Zweierkomplementzahl mit 2 durch. Das Vorzeichenbit b_7 wird nach b_6 geschoben, bleibt aber selbst unverändert. Dadurch ist sichergestellt, daß das Vorzeichen vor und nach der Division gleichbleibt (signed shift). Dieses Phänomen wurde bereits in Kapitel 2 bei dem Verschiebebefehl „arithmetisch rechts schieben" als Bedingung für die Division durch 2 genannt.

ASR führt eine Division einer Zweierkomplementzahl mit 2 durch. Das Vorzeichenbit wird festgehalten und bei jedem Verschieben zurückgeschrieben (signed shift). Eine Überschreitung des Zahlenbereiches wird in *beiden* Fällen angezeigt nach der Formel[1]):

$$\text{Overflow: } V = (N \veebar C)$$

$$\text{Carry: } C = \begin{array}{l}\text{Bit 7 bei ASL}\\ \text{Bit 0 bei ASR}\end{array}$$

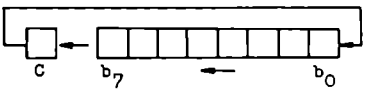
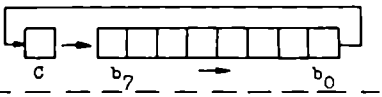
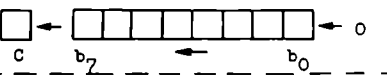
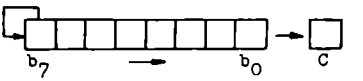
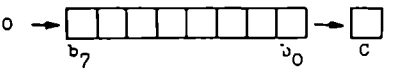
V wird gesetzt, nachdem ASL/ASR ausgeführt wurde.

Dabei sind N und C die Werte im Statusregister, die sich nach Ausführung von ASL/ASR ergeben.

Neben diesen arithmetischen Operationen lassen sich mit diesen beiden Befehlen auch Tests bestimmter Bits ausführen, nur mit der Einschränkung, daß die getesteten Bits im Gegensatz zu ROL, ROR nach einer nochmaligen Verschiebung verlorengehen, d.h. überschrieben werden (vgl. Tabelle 4.6).

[1]) Der * aus Tabelle 4.6 wird durch die Formel für V erklärt.

Tabelle 4.6: Verschiebebefehle

Mnemonic	Funktionsbeschreibung	Statusregister H I N Z V C
ROL ROLA ROLB	Rotate left M, ACCA, ACCB 	- - ↕ ↕ • ↕ - - ↕ ↕ • ↕ - - ↕ ↕ • ↕
ROR RORA RORB	Rotate right M, ACCA, ACCB 	- - ↕ ↕ • ↕ - - ↕ ↕ • ↕ - - ↕ ↕ • ↕
ASL ASLA ASLB	Arithmetic shift left M, ACCA, ACCB 	- - ↕ ↕ • ↕ - - ↕ ↕ • ↕ - - ↕ ↕ • ↕
ASR ASRA ASRB	Arithmetic shift right M, ACCA, ACCB 	- - ↕ ↕ • ↕ - - ↕ ↕ • ↕ - - ↕ ↕ • ↕
LSR LSRA LSRB	Logical shift right M, ACCA, ACCB 	- - 0 ↕ • ↕ - - 0 ↕ • ↕ - - 0 ↕ • ↕

Der logische Verschiebebefehl

LSR ist das Analogon zum ASL-Befehl. Er kann bei der Division von solchen Dualzahlen verwendet werden, die kein Vorzeichenbit besitzen. Ansonsten liegt ihre Anwendung auf demselben Gebiet wie bei ASL (Bittest).

4.7 Registerbefehle

Unter dem Begriff Registerbefehle sind alle Befehle zusammengefaßt, die sich auf das Indexregister (IX), das Adreßregister (RO) und den Stack Pointer (SP) beziehen.

Tabelle 4.7 veranschaulicht die Möglichkeiten, diese Register vom Speicher aus zu laden und in denselben zurückzuschreiben. Da alle diese Register 16 Bit breit sind, der

Speicher für diesen hypothetischen Mikroprozessor aber byte-organisiert ist (1 Byte = 8 Bits), bedingt das Laden und Abspeichern der Register einen zweimaligen Zugriff zum Speicher. Weiterhin können die Register um Eins erhöht bzw. erniedrigt werden.

Tabelle 4.7: Registerbefehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
CPX	Compare IX	$<IX_L>-<M+1>$; $<IX_H>-<M>$	- - ↕ ↕ ↕ -
DEX	Decrement IX	<IX >-1 → IX	- - - ↕ - -
DES	Decrement SP	<SP >-1 → SP	- - - - - -
DER	Decrement RO	<RO >-1 → RO	- - - ↕ - -
INX	Increment IX	<IX >+1 → IX	- - - ↕ - -
INS	Increment SP	<SP >+1 → SP	- - - - - -
INR	Increment RO	<RO >+1 → RO	- - - ↕ - -
LDX	Load IX	$<M> \rightarrow IX_H$; $<M+1> \rightarrow IX_L$	- - ↕ ↕ 0 -
LDS	Load SP	$<M> \rightarrow SP_H$; $<M+1> \rightarrow SP_L$	- - ↕ ↕ 0 -
LDR	Load RO	$<M> \rightarrow RO_H$; $<M+1> \rightarrow RO_1$	- - ↕ ↕ 0 -
STX	Store IX	$<IX_H> \rightarrow M$; $<IX_L> \rightarrow M+1$	- - ↕ ↕ 0 -
STS	Store SP	$<SP_H> \rightarrow M$; $<SP_L> \rightarrow M+1$	- - ↕ ↕ 0 -
STR	Store RO	$<RO_H> \rightarrow M$ $<RO_L> \rightarrow M+1$	- - ↕ ↕ 0 -
TXS	Transfer IX to SP	<IX >-1 → SP	- - - - - -
TSX	Transfer SP to IX	<SP >+1 → IX	- - - - - -
PSHA	Push data onto stack	<ACCA > → <SP > <SP >-1 → SP	- - - - - -
PSHB		<ACCB > → <SP > <SP >-1 → SP	- - - - - -
PULA	Pull data from stack	<SP >+1 → SP <<SP >> → ACCA	- - - - - -
PULB		<SP >+1 → SP <<SP >> → ACCB	- - - - - -

Indexregister (IX), Adreßregister (RO) und Stack Pointer (SP)

Wichtige Befehle stellen INX bzw. DEX dar, da sie das Zero-Flag beeinflussen. Dies ermöglicht die Verwendung des Indexregisters außer zu den behandelten Adressierungszwecken (Kapitel 3) auch zur Kontrolle von Programmschleifen (Abschnitt 4.8). Das Indexregister wird als Zähler verwendet, dessen Ausgangszustand der gewünschten Anzahl n von Schleifendurchgängen entspricht. Hat der Zähler nach n Durchgängen (n-mal DEX!) den Stand Null erreicht, so kann über einen bedingten Sprung (Z = 1;?) die Schleife verlassen werden.

Für den betrachteten hypothetischen Mikroprozessor ist der Stack in Form eines Software- oder Pointer-Stack realisiert. Der Stack (Stapel engl.: stack) befindet sich in einem RAM außerhalb der CPU. Zur Kontrolle des Stacks steht dem Programmierer der Stack Pointer (SP) zur Verfügung. Mit ihm ist er in der Lage, die Anfangsadresse des als Stack definierten RAM-Bereiches festzulegen, und zwar irgendwo innerhalb des verfügbaren Speicherraumes. Hieraus erkennt man, wie wichtig es ist, daß der Stack Pointer dieselbe Bitbreite besitzt wie der Programmzähler. Jedoch muß einschränkend bemerkt werden, daß der Stack so gelegt werden sollte, daß er nicht in den Bereich des Benutzerprogramms hineinwächst, wo zu verarbeitende Daten abgelegt sind. Der Stack „wächst" nun je nach Mikroprozessortyp zu höheren oder niedri-

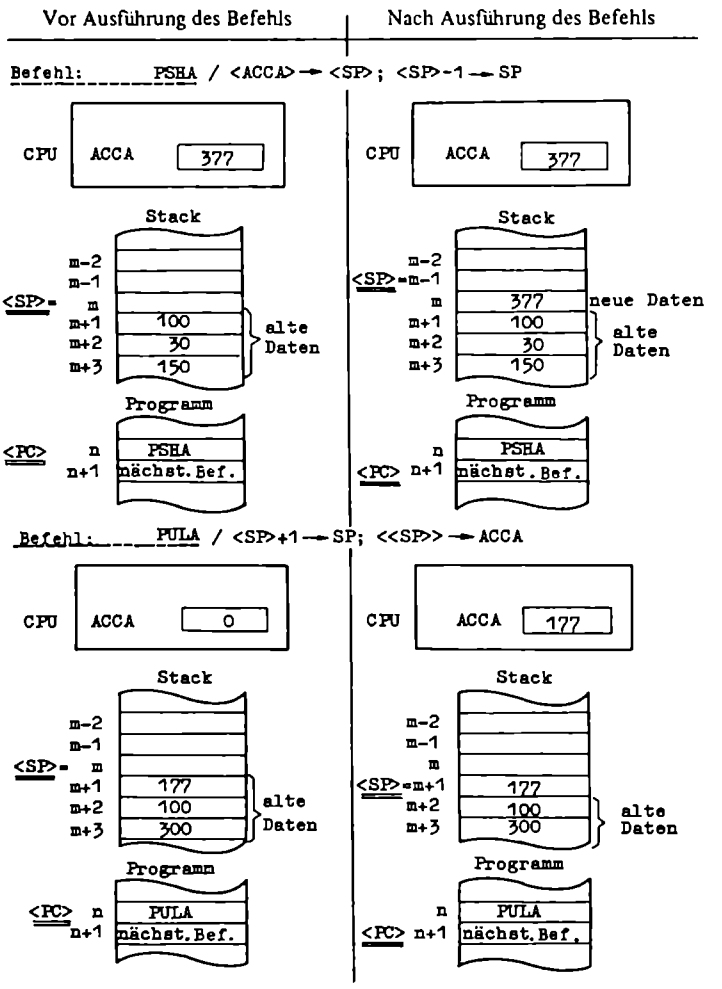


Bild 4.3: Die Befehle PUSH und PULL

geren Speicheradressen. Der Datenaustausch (Transfer) zwischen den Akkumulatoren A und B sowie dem Stack erfolgt mit den Befehlen PSHA bzw. PSHB und PULA bzw. PULB.

Die Befehle PSHX bzw. PULX ermöglichen dem Programmierer, den Akkumulatorinhalt auf den Stack zu schreiben und zurückzulesen. Der Stack Pointer übernimmt dabei die Kontrolle der laufenden Adresse des Stacks, denn mit PSHX- und PULX-Befehlen ist eine automatische Erniedrigung bzw. Erhöhung des Stack Pointers verbunden.

PSHX beinhaltet die Adressierungsart *autodecrement* (postdecrement), PULX dagegen *autoincrement* (preincrement).

Die genaue Funktion von PUSH und PULL ist aus Bild 4.3 zu entnehmen. Dabei beachte man die Stellung des Stack Pointers jeweils vor und nach dem PUSH- bzw. PULL-Befehl.

Diese Befehle ermöglichen aufgrund ihrer Adressierungsart nur den Zugriff zu benachbarten Speicherplätzen im Stack. Dies wird manchmal als Einschränkung empfunden und kann mit den Befehlen LDS, INS und DES sowie TSX und TXS dadurch umgangen werden, daß der Stack Pointer neu geladen (initialisiert) wird. Die beiden letzten Befehle sind für Anwendungen wichtig, bei denen auf einfache Weise ein wahlfreier Zugriff zum Stack realisiert werden soll. Dabei gilt es zu bedenken, daß ein mehrmaliges Anwenden von TSX und TXS bei gleichzeitiger Veränderung des Indexregisters die durch PUSH und PULL selbständig erfolgte Kontrolle des Datentransfers zwischen Stack und CPU zunichte macht. Der Anwender muß sich beim erneuten Laden des Stack Pointers überlegen, zu welcher Stelle im Stack er Zugriff haben möchte. Diese Überlegung war bei PUSH und PULL nicht notwendig, da beide Befehle den Stack Pointer durch die Adressierungsarten *autoincrement* und *autodecrement* selbständig steuern.

Zusammenfassend kann gesagt werden, daß mit dem Stack und den ihn steuernden Befehlen Hilfsmittel zur Verfügung stehen, die es dem Programmierer gestatten, ohne große Schwierigkeiten zu einer übersichtlichen, effektiven und gut zu testenden Software zu gelangen.

4.8 Verzweigungs- und Sprungbefehle

Alle bisher vorgestellten Befehle besaßen nicht die Möglichkeit, den Programmablauf abhängig von einer Bedingung (z.B. Carry-Flag gesetzt, ⟨ACCA⟩ = 0, ⟨M⟩ ⩽ 0) zu beeinflussen. Die in Tabelle 4.8 zusammengestellten Befehle ermöglichen eine Programmverzweigung, d.h. das Programm wird, wenn der Test positiv ausgefallen ist, an einer anderen Stelle im Speicher fortgesetzt. Dabei werden zwei Arten von Verzweigungen unterschieden:

1. Die unbedingten Verzweigungen (unconditional branches),
2. die bedingten Verzweigungen (conditional branches).

Die erste Art wird deshalb als „unbedingt" bezeichnet, weil in diesem Fall eine Verzweigung immer durchgeführt wird (BRA).

Bei der zweiten Art hingegen wird die Durchführung einer Verzweigung von einer Bedingung abhängig gemacht. Diese wird im Mnemonic des jeweiligen Befehls zum Ausdruck gebracht (z.B. BCC branch if carry clear).

Bei den bedingten Verzweigungen fällt auf, daß als Bedingung nicht nur ein einzelnes Bit des Statusregisters herangezogen werden kann, sondern auch deren Verknüpfungen (z.B. C ∨ Z).

Der BRA- und der JMP-Befehl entsprechen einander funktionsmäßig, nur daß der BRA-Befehl eine andere Adressierungsart verwendet, nämlich relative Adressierung, im Gegensatz zur extended und indexed Adressierung, die bei JMP verwendet wird.

Im weiteren sollen die bedingten Verzweigungsbefehle untersucht werden. Dazu sei noch einmal auf das Statusregister mit seinen Flags zurückgegriffen.

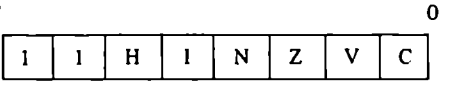


Bild 4.4: Statusregister

Folgende Einteilung bietet sich zur Klassifizierung der Verzweigungsbefehle an: Die Befehle werden eingeteilt in solche, die sich auf ein einzelnes Flag beziehen, und solche, bei denen Flags zur Definition der Bedingung untereinander logisch verknüpft sind.

Einzelne Flags, auf die beim Test Bezug genommen wird, sind:

	Verzweigung, wenn Flag = 1 bei	Verzweigung, wenn Flag = 0 bei
Carry-Flag:	BCS	BCC
Overflow-Flag:	BVS	BVC
Zero-Flag:	BEQ	BNE
Negative-Flag:	BMI	BPL

Aus der obigen Aufstellung wird ersichtlich, daß diese Befehle immer paarweise zusammengefaßt werden können und daß sie zueinander hinsichtlich der Sprungbedingung komplementär sind. Es wird verzweigt, sobald im Falle des Sprungbefehls BCS die Bedingung Carry = 1 erfüllt ist, bei BCC jedoch nur, wenn das Carry-Flag = 0 ist.

Sehr schön läßt sich an dieser Stelle der Nutzen der Bittest- und Vergleichsbefehle erkennen, die allesamt das C-, V-, Z- und N-Flag beeinflussen. Abhängig vom jeweiligen Ergebnis läßt sich eine Verzweigung durch Abfragen des betreffenden Flags des Statusregisters erreichen.

Auf Verknüpfungen einzelner Flags untereinander greift die zweite Gruppe zurück. Dabei handelt es sich um die Paare:

	Verknüpfung = 1	Verknüpfung = 0
C ∨ Z	BLS	BHI
N ⊻ V	BLT	BGE
Z ∨ (N ⊻ V)	BLE	BGT

BHI und BLS können verwendet werden, um unmittelbar nach dem CMP-Befehl zu testen, ob der Akkumulatorinhalt größer, kleiner oder gleich dem des adressierten Operanden war. Hierbei hat man es mit Dualzahlen ohne Vorzeichen zu tun (unsigned binary numbers), d.h. es werden die entsprechenden Bits logisch miteinander verglichen.

Wurde der CMP-Befehl im Zusammenhang mit Zweierkomplementzahlen verwendet, so müssen für anschließende Entscheidungen (z.B. ≥ 0; < 0; > 0; ≤ 0) die Befehle BGE, BLT, BGT und BLE herangezogen werden. BGE und BLT sind mit dem N- und V-Flag verbunden (N $\veebar$ V).

BGE und BLT werden verwendet, um anschließend an Rechenoperationen mit Zweierkomplementzahlen nachzuprüfen, ob ein Ergebnis ≥ 0 oder < 0 ist. Befehle, die das N- und V-Flag beeinflussen, wie z.B. Arithmetik-(ADDA), Vergleichs-(CMP) und Verschiebeoperationen (ROR), lassen eine Prüfung des Ergebnisses mit BGE oder BLT zu. Aus der obigen Aufstellung erkennt man, daß die beiden Befehle BGE und BLT zueinander komplementär sind.

Die beiden restlichen Befehle BGT und BLE ergänzen die beiden oben behandelten derart, daß das Z-Flag mit der für BGE bzw. BLT maßgebenden logischen Verknüpfung N $\veebar$ V Oder-verknüpft wird.

BGT entspricht somit BGE mit einem zusätzlichen Test auf $\neq 0$ und BLE entspricht BLT mit einem zusätzlichen Test auf 0.

Tabelle 4.8: Verzweigungs- und Sprungbefehle

Mnemonic	Funktionsbeschreibung		Statusregister H I N Z V C
BRA	Branch always		- - - - - -
BEQ	Branch if = 0	if Z=1	- - - - - -
BGE	Branch if ≥ 0	if N ⊻ V=0	- - - - - -
BGT	Branch if > 0	if Z ∨ (N ⊻ V)=0	- - - - - -
BHI	Branch if higher	if C ∨ Z=0	- - - - - -
BLE	Branch if ≤ 0	if Z ∨ (N ⊻ V)=1	- - - - - -
BLS	Branch if lower or same	if C ∨ Z=1	- - - - - -
BLT	Branch if < 0	if N ⊻ V=1	- - - - - -
BMI	Branch if minus	if N=1	- - - - - -
BNE	Branch if ≠ 0	if Z=0	- - - - - -
BPL	Branch if plus	if N=0	- - - - - -
BCC	Branch if carry clear	if C=0	- - - - - -
BCS	Branch if carry set	if C=1	- - - - - -
BVC	Branch if overflow clear	if V=0	- - - - - -
BVS	Branch if overflow set	if V=1	- - - - - -
JMP	Jump		- - - - - -

4.9 Anruf- und Rücksprungbefehle für Unterprogramme

In diesem Abschnitt werden die in Tabelle 4.9 zusammengestellten Befehle

BSR, JSR und RTS

behandelt.

Tabelle 4.9: Anruf- und Rücksprungbefehle für Unterprogramme

Mnemonic	Funktionsbeschreibung	Statusregister H I N Z V C
BSR	Branch to subroutine	- - - - - -
JSR	Jump to subroutine	- - - - - -
RTS	Return from subroutine	- - - - - -

Das Bestreben eines Programmierers sollte es sein, ein Programm zu erstellen, das als ganzes eine gestellte Aufgabe löst. Gleichzeitig sollte es kleine Teilabschnitte, sogenannte Unterprogramme (subroutines), enthalten, die nicht nur für das eine spezielle Problem geeignet, sondern vielseitig verwendbar sind. Er sollte bestrebt sein, sein Programm modular aufzubauen. Man denke an Eingabe- und Ausgaberoutinen, Prüfroutinen verschiedenster Art und an Algorithmen (z.B. Multiplikation, Division), die nicht nur von einem einzigen Programmierer benötigt werden, um ein Problem zu lösen. Diese Unterroutinen bilden häufig das Grundgerüst für weitere Programme. Dazu ist natürlich eine Voraussetzung, daß die Dokumentation über die Unterroutine eine genaue Definition ihrer Funktion enthält. Eine Unterroutine benötigt ihre Operanden vom Hauptprogramm. Des weiteren muß sie wissen, wohin Ergebnisse auszugeben sind . (Die von der Unterroutine benötigten Operanden werden allgemein als Argumente bezeichnet.) Ein Programmteil, der dies in der Unterroutine leistet, ist der sogenannte Argumenten- und Ergebnistransfer. Häufig ist es nützlich zu wissen, welchen Zustand die von der Unterroutine benötigten Register zu Beginn und am Ende haben, ob z.B. der Akkumulator A in einem genau definierten logischen Zustand gelassen wird oder nicht. Eine Unterroutine kann, vorausgesetzt sie besitzt obige Angaben, vom Anwender eingesetzt werden, ohne daß er im einzelnen genau weiß, wie die Unterroutine arbeitet. Der Anwender hat auf diese Weise die Möglichkeit, sich aus Moduln sein Programm aufzubauen und diese mittels eines Hauptprogramms zu verknüpfen..

Prinzipiell gibt es nun zwei Möglichkeiten, Unterprogramme in ein Programm einzubauen. Die erste und naheliegende ist die, eine Unterroutine einmal im Speicher abzulegen und sie durch das Hauptprogramm anzurufen, wann immer sie benötigt wird. Diese Anordnung der Unterroutine wird auch als closed subroutine bezeichnet (Bild 4.5).

Neben dem Vorteil des geringen Speicherplatzbedarfs für die Unterroutine muß ein Zeitverlust in Kauf genommen werden, der sich aus der Zeit für den Anruf (JSR bzw. BSR) und den Rücksprung (RTS) zusammensetzt.

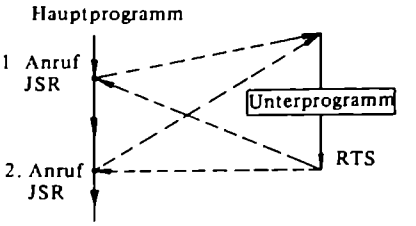


Bild 4.5: Geschlossene Unterroutine (closed subroutine)

Die zweite Möglichkeit eine Unterroutine anzuordnen ist in Bild 4.6 gezeigt. Hierbei wird die Unterroutine genauso oft im Speicher abgelegt, wie das Hauptprogramm auf sie zurückgreift. Dies setzt natürlich eine Abänderung der Unterroutine gegenüber der obigen geschlossenen Form voraus (Argumententransfer geändert!). Der Zeitverlust bei der ersten Möglichkeit wird eingetauscht mit einem größeren Speicherplatzbedarf. Auch geht mit dieser Darstellung der modulare Aufbau des Programms etwas verloren. Die Unterroutine ist ins Hauptprogramm integriert.

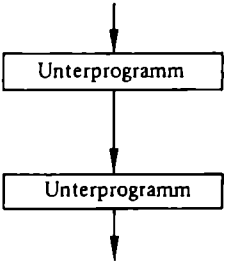


Bild 4.6: Offene Unterroutine (open subroutine)

Im Zusammenhang mit der in Bild 4.5 vorgestellten „closed subroutine" ergeben sich folgende Fragen: Wie wird eine Unterroutine angerufen, in welcher Weise werden die Argumente von und zur Unterroutine transferiert und wie erfolgt die Rückkehr aus der Unterroutine zum Hauptprogramm?

Benötigt das Hauptprogramm eine Unterroutine, so erfolgt deren Anruf mit dem Befehl JSR bzw. BSR. JSR und BSR unterscheiden sich nur in der Adressierungsart, sonst leisten beide dasselbe. Mit der Ausführung des Befehls JSR wird dafür gesorgt, daß nach Beendigung der Unterroutine eine Rückkehr zu der Speicheradresse ermöglicht wird, von der aus der Anruf erfolgte. Mit JSR wird zur Unterroutine gesprungen und gleichzeitig die Rückkehradresse auf dem Stack abgespeichert, d.h. JSR führt zusätzlich zum Sprung noch einen PUSH-Befehl aus.

Die Rückkehradresse ist dabei die Adresse des auf den JSR-Befehl folgenden Befehls (s. Beispiel Bild 4.7). Anders ausgedrückt bedeutet dies: $\langle PC \rangle + 3 \rightarrow \langle SP \rangle$, $\langle SP \rangle - 1$; $\langle SP \rangle - 2 \rightarrow SP$. Der Stack wächst zu niedrigeren Speicheradressen. Die Abspeicherung der Rückkehradresse auf dem Stack ist bei vielen Mikroprozessoren üblich.

Andere Möglichkeiten wären:

– Ablegen der Rückkehradresse in einem internen CPU-Register (= subroutine linkage register; s. Beispiel Bild 4.8).

– Ablegen der Rückkehradresse in der ersten Speicherzelle des Unterprogramms.

Letztere Methode wird bei Mikroprozessoren nicht angewendet, da Unterroutinen meist in ROMs gespeichert sind, in die bekanntermaßen nicht geschrieben werden kann.

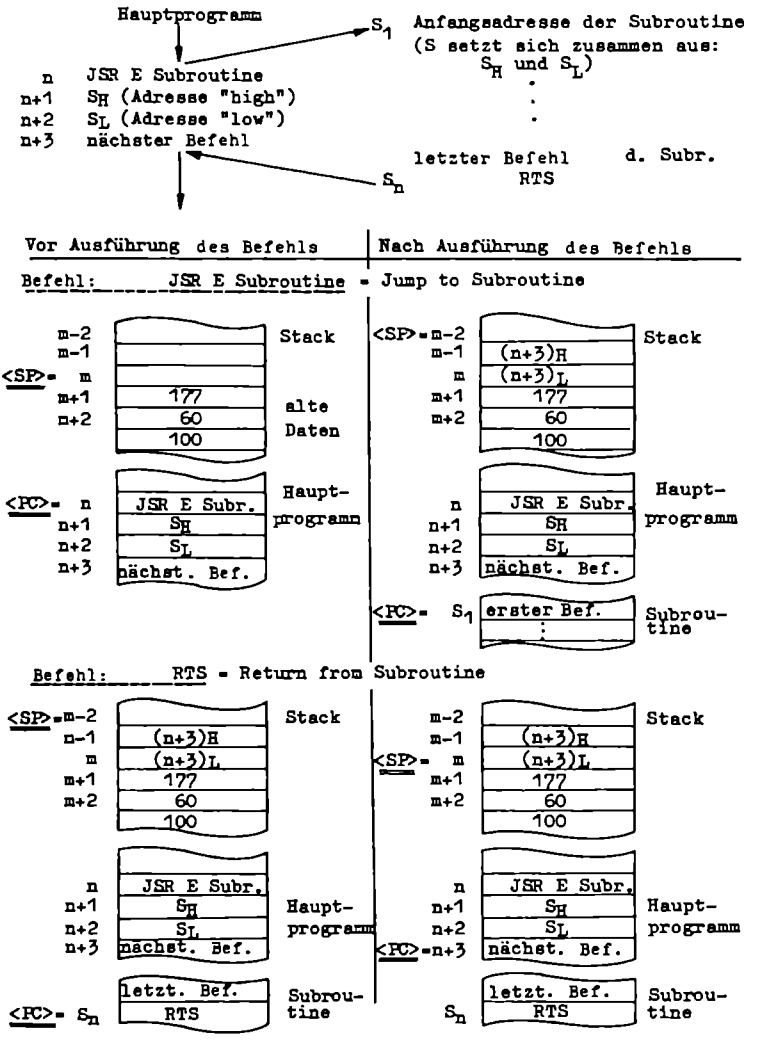


Bild 4.7: Die Befehle JSR und RTS

Um den Rücksprung aus der Unterroutine zum Hauptprogramm zu ermöglichen, wird diese mit dem Befehl RTS abgeschlossen. Dies bewirkt mit der Ausführung von RTS das Laden des Programmzählers (PC) mit der Rückkehradresse (⟨⟨SP⟩⟩, ⟨⟨SP⟩ + 1⟩ → PC) (s. Beispiel in Bild 4.7).

4.10 Einfluß der Stackarchitektur auf den Argumententransfer

Bild 4.8 zeigt mögliche Unterroutinenanrufe am Beispiel einer Multiplikationsunterroutine für 8-Bit-Dualzahlen. Mit diesem Beispiel soll gezeigt werden, wie wichtig eine bestimmte Stackarchitektur und wie nützlich bestimmte Adressierungsarten sind, um einen Argumententransfer zur Unterroutine schnell, d.h. mit wenigen Befehlen durchzuführen. In den ersten beiden Beispielen wird von einer Stackarchitektur ausgegangen, die Eigenschaften des Hardware- und des Software-Stack vereinigt. Das Adreßregister RO wird als Eingang zum Stack benutzt. Ansonsten ändert sich nichts bezüglich der Architektur des hypothetischen Mikroprozessors. Der letzte auf dem Stack abgespeicherte Operand befindet sich in diesem Register RO. Wird abermals auf den Stack geschrieben, so wird der Inhalt von RO in den RAM-Bereich geschoben und der neue Operand wird nach RO geladen. RO wird aus diesem Grund als Verbindungsregister (linkage register) bezeichnet. Im Zusammenhang mit Unterroutineanrufen spricht man von einem „subroutine linkage register".

In Bild 4.8a erfolgt der Anruf der Unterroutine mit dem Befehl JSR RO MULT. Dabei ist MULT die Anfangsadresse der Unterroutine und damit das Sprungziel. RO bezeichnet das subroutine linkage register. Dort ist jetzt die Adresse (n + 1) abgelegt, der Speicherplatz, der das erste Argument für die Unterroutine enthält. Da auf RO die Registeradressierungsart *autoincrement* angewandt werden kann, ist es möglich, das erste Argument mit dem Befehl LDAA RO ↑ in Akkumulator A zu laden. Nach ausgeführter Operation wird ⟨RO⟩ inkrementiert, d.h. RO enthält jetzt die Adresse (n + 2) des zweiten Arguments. Dieses wird ebenfalls von der Unterroutine eingelesen. Damit befinden sich beide Operanden in der Unterroutine und die Multiplikation kann erfolgen. Am Ende der Berechnung wird das Ergebnis an die Adressen (n + 3) und (n + 4) ausgegeben. Danach erfolgt der Rücksprung zum Hauptprogramm an die Adresse (n + 5). An diesem Beispiel in Bild 4.8a ist gezeigt, wie nützlich und vorteilhaft die Adressierungsart *autoincrement* eingesetzt werden kann, weil der korrekte Rücksprung zum Hauptprogramm eine Erhöhung der Rückkehradresse um Eins bedingt, sobald ein Argument von oder zur Unterroutine transferiert wird. Der Argumententransfer wird entscheidend von den vorhandenen Adressierungsarten und der Stackarchitektur beeinflußt. Bild 4.8b liegt dasselbe Prinzip zugrunde, nur daß anstelle der Operanden deren Adressen angegeben sind. Dies erfordert die Adressierungsart *indirect autoincrement*.

Besitzt ein Mikroprozessor nicht die genannte Stackarchitektur und kann er nicht auf die Adressierungsarten wie *autoincrement*, *indexed* und *indirect* zurückgreifen, so müssen die Operanden in Registern oder in speziellen, nur für die Unterroutine bestimmte Speicherplätze abgelegt sein, bevor deren Anruf über JSR MULT erfolgt (Bild 4.9).

```
        /  Hauptprogramm
  n        JSR RO MULT
n+1        Argument 1
n+2        Argument 2
n+3        Argument 3
n+4        Argument 4
n+5        Main routine        / Fortsetzung des Hauptprogramms


        /  Subroutine MULT
        /  1. Argument: Multiplikand
        /  2. Argument: Multiplikator
        /  3. Argument: Produkt high
        /  4. Argument: Produkt low

                                                                  ⟨RO⟩
                                                                    ↓
                                / Inhalt von RO nach Ausführung eines Befehls   [n+1]
                                  (z.B. JSR, LDAA)

MULT,      LDAA RO↑             / Argument 1 in ACCA laden                      [n+2]

           LDAB RO↑             / Argument 2 in ACCB laden                      [n+3]

                                / Multiplikationsroutine:
                                / ⟨ACCA⟩ · ⟨ACCB⟩
                                / Ergebnis:
                                / obere 8 Bits des Produkts
                                / in PRODH
                                / untere 8 Bits des Produkts
                                / in PRODL

           LDAA E  PRODH        / Produkt high in ACCA laden

           STAA RO↑             / ⟨ACCA⟩ nach Argument 3 abspeichern            [n+4]

           LDAA E  PRODL        / Produkt low in ACCA laden

           STAA RO↑             / ⟨ACCA⟩ nach Argument 4 abspeichern            [n+5]

           RTS  RO              / zurück zum Hauptprogramm
```

Bild 4.8a: Argumententransfer: *Autoincrement Adressierung (RO)*
(Befehlsvorrat des hypothetischen Mikroprozessors)

```
        /  Hauptprogramm
   n        JSR RO MULT
 n+1        Argument 1
 n+2        Argument 2
 n+3        Argument 3
 n+4        Argument 4
 n+5        Main routine        / Fortsetzung des Hauptprogramms


        /  Subroutine MULT
        /  1. Argument: Adresse des Multiplikanden
        /  2. Argument: Adresse des Multiplikators
        /  3. Argument: Adresse des Produkts high
        /  4. Argument: Adresse des Produkts low
```

```
                        / Inhalt von RO nach Ausführung eines Befehls
                          (z.B. JSR, LDAA)
MULT,   LDAA IRO↑       / Argument 1 in ACCA laden

        LDAB IRO↑       / Argument 2 in ACCB laden

                        / Multiplikationsroutine:
                        / (ACCA) · (ACCB)
                        / Ergebnis:
                        / obere 8 Bits des Produkts
                        / in PRODH
                        / untere 8 Bits des Produkts
                        / in PRODL

        LDAA E PRODH    / Produkt high in ACCA laden

        STAA IRO↑       / (ACCA) nach Argument 3 abspeichern

        LDAA E PRODL    / Produkt low in ACCA laden

        STAA IRO↑       / (ACCA) nach Argument 4 abspeichern

        RTS  RO         / zurück zum Hauptprogramm
```

Bild 4.8b: Argumententransfer: *Indirect Autoincrement Adressierung (RO)*
(Befehlsvorrat des hypothetischen Mikroprozessors)

Kein Argumententransfer durch Subroutine möglich (siehe Stackarchitektur). Argumententransfer im Hauptprogramm (Argument 1 bis 4 sind Adressen).

```
        /  Hauptprogramm
  n     LDAA E  ARGUMENT 1
n+1     STAA E  M1              / Multiplikand nach M1
n+2     LDAA E  ARGUMENT 2
n+3     STAA E  M2              / Multiplikator nach M2
n+4     JSR  E  MULT            / Anruf der Multiplikationssubroutine
                                / zweier Dualzahlen (8-Bit)
n+5     LDAA E  PRODH
n+6     STAA E  ARGUMENT 3      /Produkt high nach Argument 4 abspeichern
n+7     LDAA E  PRODL
n+8     STAA E  ARGUMENT 4      / Produkt low nach Argument 3 abspeichern
n+9     Main routine            / Fortsetzung des Hauptprogramms
                .
                .
                .
ARGUMENT 1,     Multiplikand
ARUGMENT 2,     Multiplikator
ARGUMENT 3,     0
ARGUMENT 4,     0

        /  Subroutine MULT
        /  Multiplikand in M1
        /  Multiplikator in M2
        /  Produkt high in PRODH
        /  Produkt low in PRODL

MULT,                   / Multiplikation zweier (8-Bit) Dualzahlen
                        / (M1) · (M2)
                        / Ergebnis:
        .               / obere 8 Bits des Produkts in PRODH
        .               / untere 8 Bits des Produkts in PRODL
        RTS             / zurück zum Hauptprogramm
```

Bild 4.9: Befehlsvorrat des hypothetischen Mikroprozessors *ohne* Verwendung von RO

4.11 Ergänzungen zum Befehlsvorrat

Bei der Betrachtung des Befehlsvorrats fällt auf, daß es keine typischen Eingabe- bzw. Ausgabebefehle gibt. Dies hat seinen Grund darin, daß der vorliegende hypothetische Mikroprozessor jegliche Eingabe und Ausgabe über die beiden Befehle Load und Store bewerkstelligt. Bei anderen Mikroprozessoren finden sich Eingabe- und Ausgabebefehle der Form IN bzw. OUT, mit denen die Ein- und Ausgabe von Daten zu externen Geräten erfolgt.

Der Befehl NOP (no operation) erhöht den Programmzähler um Eins. Dieser Befehl wird in Programmentwicklungsphasen häufig verwendet, um an kritischen Stellen Platz für etwaige spätere Korrekturen zu haben.

Der vorliegende Befehlsvorrat weist bei weitem nicht alle möglichen Befehle auf. Es werden vor allem solche vermißt, die eine Arithmetik mit Doppelgenauigkeitszahlen ermöglichen, wie z.B.:

Doppelgenauigkeit: Addition (Double Add),
Subtraktion (Double Subtract),
Verschieben (Long Shift).

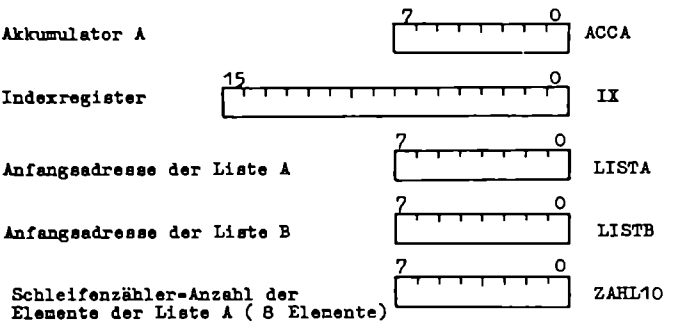
Auch fehlen, um nur einige zu nennen, Befehle wie Multiplizieren und Dividieren sowie Befehle, die bestimmte Bits oder Bytes der Akkumulatoren verändern, wie z.B. Lösche viertes Bit des Akkumulators A (Byte-, Bitmanipulation). Natürlich liegt dies auch in der Architektur des gewählten Prozessors begründet (8-Bit-Mikroprozessor).

Eine Stärke dieses hypothetischen Mikroprozessors sind seine Adressierungsarten (Bild 4.2c). Diese gestatten, wie im letzten Abschnitt an einigen Programmbeispielen gezeigt wird, eine effektive Programmierung.

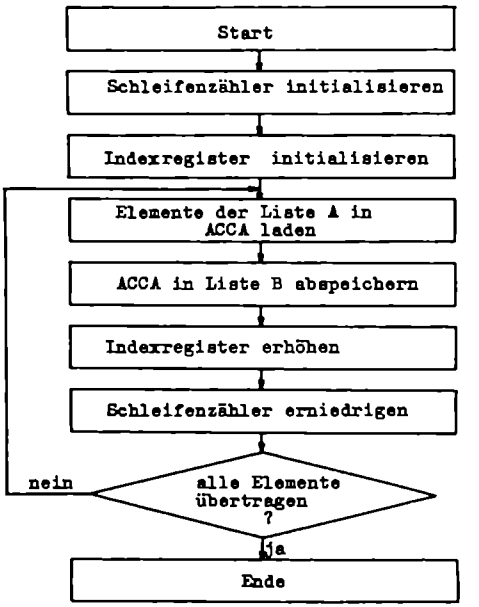
4.12 Beispiele zum Befehlsvorrat und zu den Adressierungsarten

Beispiel zur Indexed Adressierung

Elemente einer Liste A in Liste B übertragen



Flußdiagramm



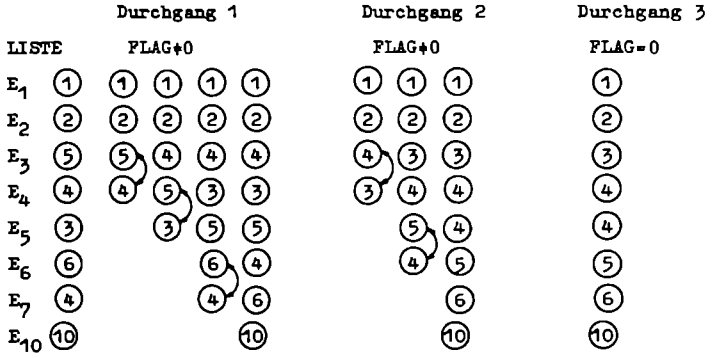
Beispiel zur Indexed Adressierung

Routine zur Übertragung von Elementen einer Liste A in eine Liste B

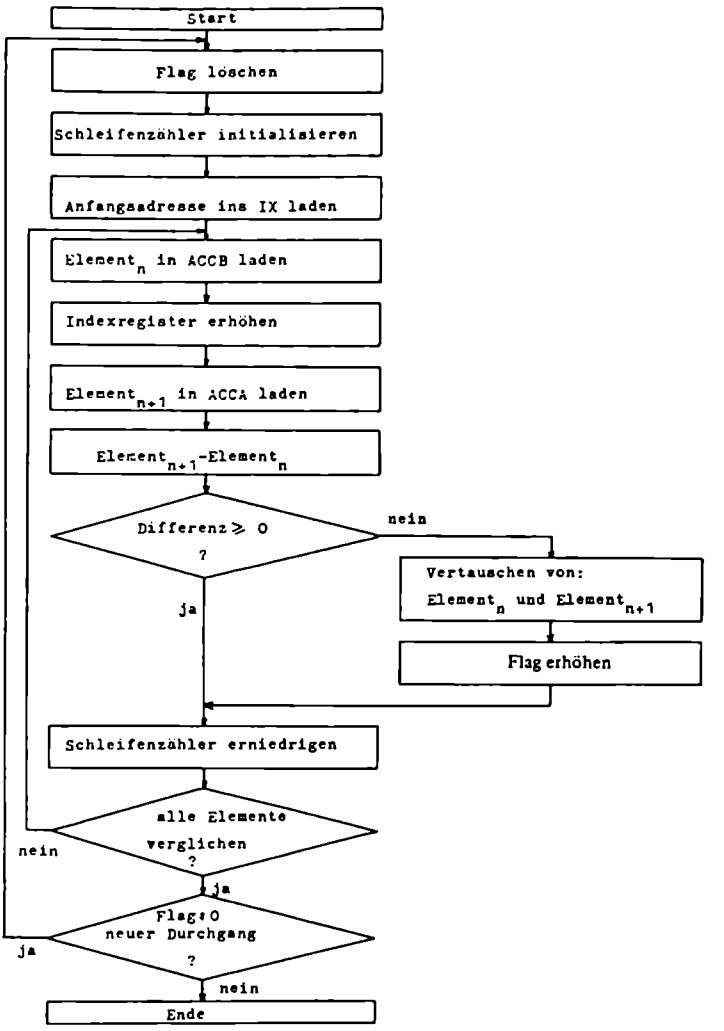
```
                LDAA # 10       /Anzahl der Elemente in ACCA laden
                                /10_8 → ACCA
                STAA E ZAHL10   /Schleifenzähler initialisieren
                                /⟨ACCA⟩ → ZAHL10
                LDX # 0         /Indexregister initialisieren
                                /0 → IX
UBTRAG,         LDAA X LISTA    /Element der Liste A in ACCA laden
                                /⟨LISTA + ⟨IX⟩⟩ → ACCA
                STAA X LISTB    /ACCA in Liste B speichern
                                /⟨ACCA⟩ → LISTB + ⟨IX⟩
                INX             /Erhöhen des IX um 1
                DEC E ZAHL10    /Schleifenzähler erniedrigen
                BGT UBTRAG      /alle Elemente übertragen?
                                /nein, zurück zu UBTRAG
ENDE,           BRA ENDE        /ja, Programmende

LISTA,          E1

                .
                .
                E10
LISTB,          0
                .
                .
                .
                0
ZAHL10,         0
```

Sortieren von Elementen einer Liste (integer, unsigned)

```
Aufgabe: Betragsmäßiges Sortieren der Elemente vom kleinsten
         zum größten Wert.

Akkumulator A                                   ACCA
Akkumulator B                                   ACCB
Indexregister                                   IX
Anfangsadresse der Liste                        LISTA
Zeiger auf die Anfangsadresse                   LISTAZ
Zähler für die Anzahl der Elemente              ZAHL10
(10₈ Elemente)
FLAG≠0: wenn Elemente vertauscht                FLAG
        wurden: neuer Durchgang
FLAG=0: wenn keine Elemente vertauscht
        wurden: kein Durchgang mehr
```



Flußdiagramm zur Sortierroutine



Sortierroutine

/Hauptprogramm

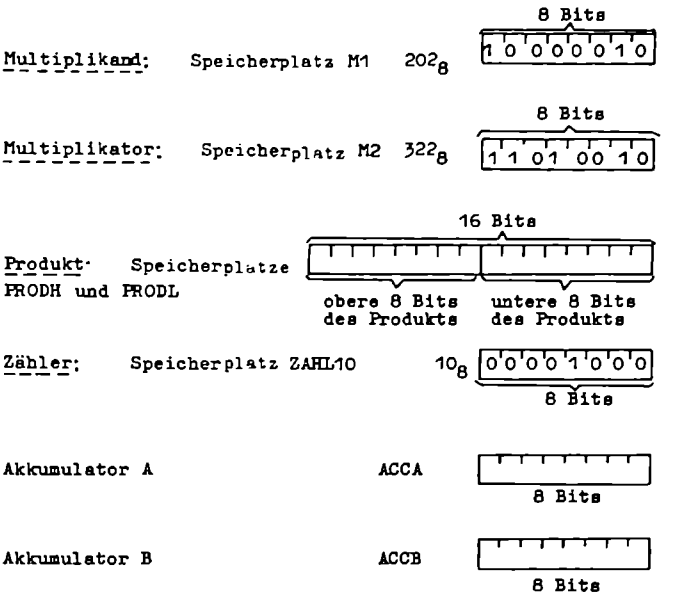
```
NEUDRG,   CLR E  FLAG      /Flag löschen
          LDAA  # 10
          STAA E  ZAHL10   /Schleifenzähler initialisieren
          LDX E  LISTAZ    /Anfangsadresse der Liste ins
                           /Indexregister laden
DIFF,     LDAB X  0        /Element_n in ACCB laden
          INX              /Indexregister erhöhen
          LDAA X  0        /Element_n+1 in ACCA laden
          SBA              /Element_n+1 – Element_n
                           /⟨ACCA⟩ – ⟨ACCB⟩→ ACCA
          BGE  NOSUB       /Differenz ≥ 0, ja, Verzweigung
                           /zu NOSUB
          JSR E  SWITCH    /nein, Sprung zur Subroutine:
                           /Vertauschen der Elemente
NOSUB,    DEC E  ZAHL10    /Schleifenzähler erniedrigen
          BGT  DIFF        /sind alle Elemente verglichen?
          TST E  FLAG      /testen des Flag auf ≠ 0
          BNE  NEUDRG      /Flag ≠ 0, neuer Durchgang
ENDE,     BRA  ENDE        /Programmende

LISTAZ,   LISTA
LISTA,    1
          2
          .
          .
          .
          10
FLAG,     0
ZAHL10,   0
```

/Subroutine: Vertauschen von zwei Elementen und abspeichern

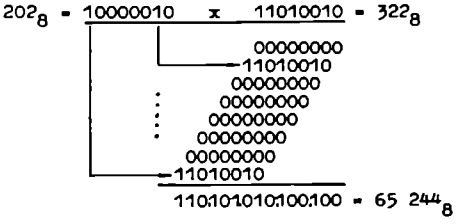
```
SWITCH,   LDAA X  0        /Element_n+1 in ACCA laden
          DEX              /Indexregister erniedrigen
          STAA X  0        /Element_n+1 nach Adresse_n abspeichern
          INX              /Indexregister erhöhen
          STAB X  0        /Element_n nach Adresse_n+1 abspeichern
          INC E  FLAG      /FLAG um Eins erhöhen
          RTS              /zurück zum Hauptprogramm
```

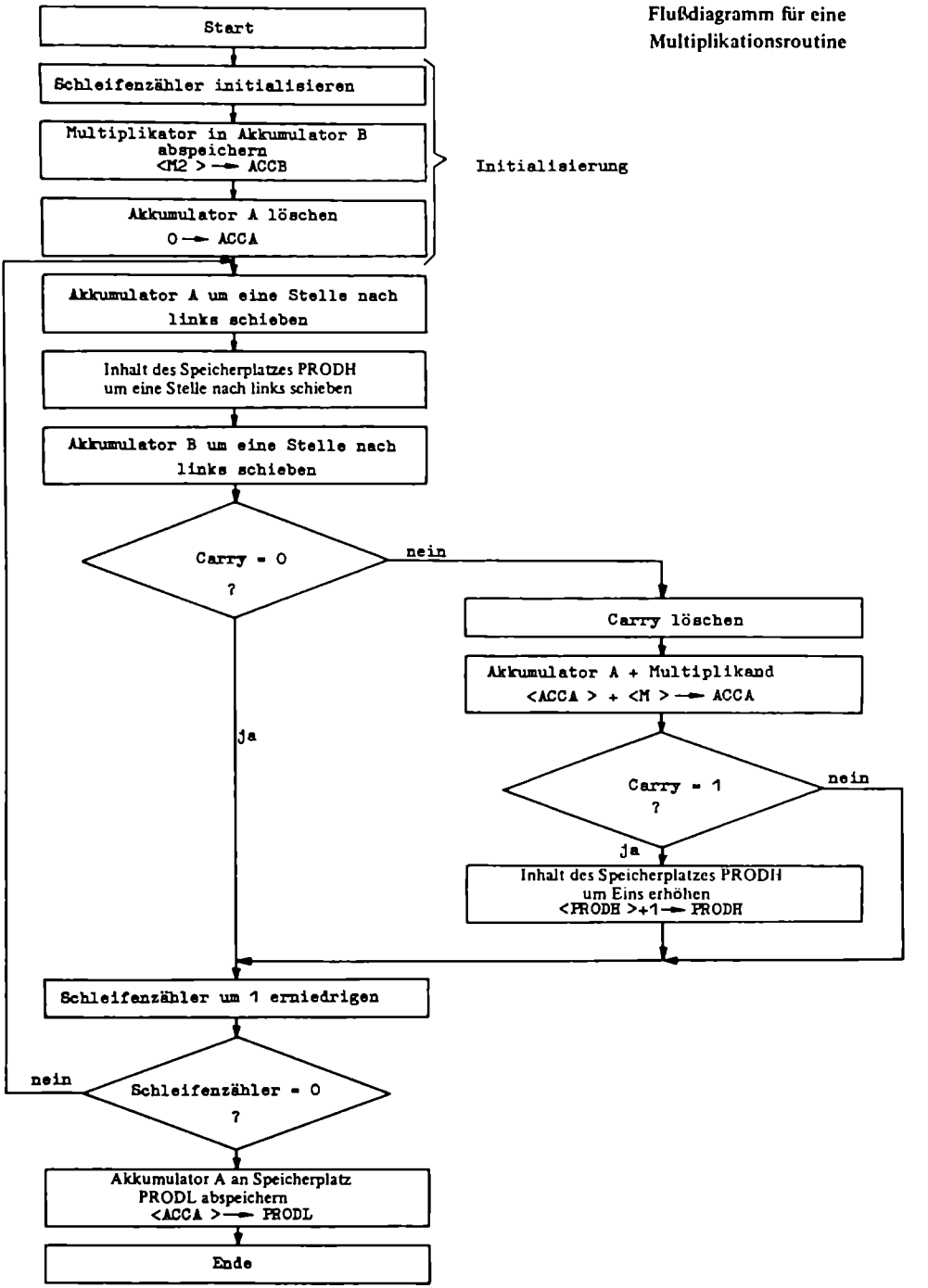
Beispiel zu den Verschiebebefehlen

Multiplikation zweier Dualzahlen
(integer, unsigned)



Beispiel einer dualen Multiplikation:



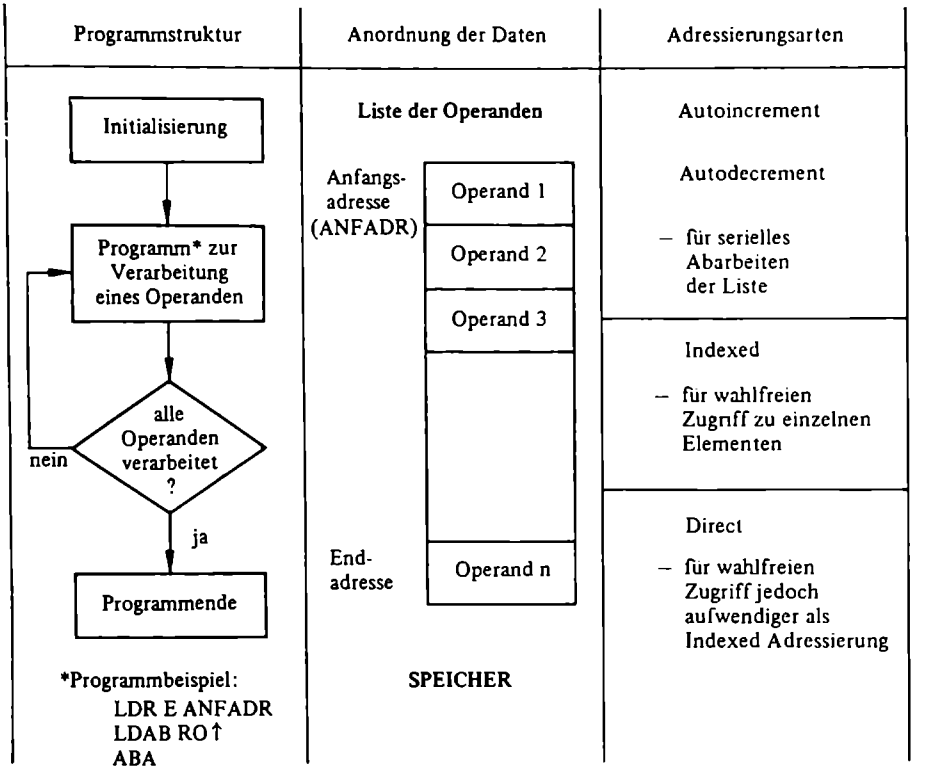


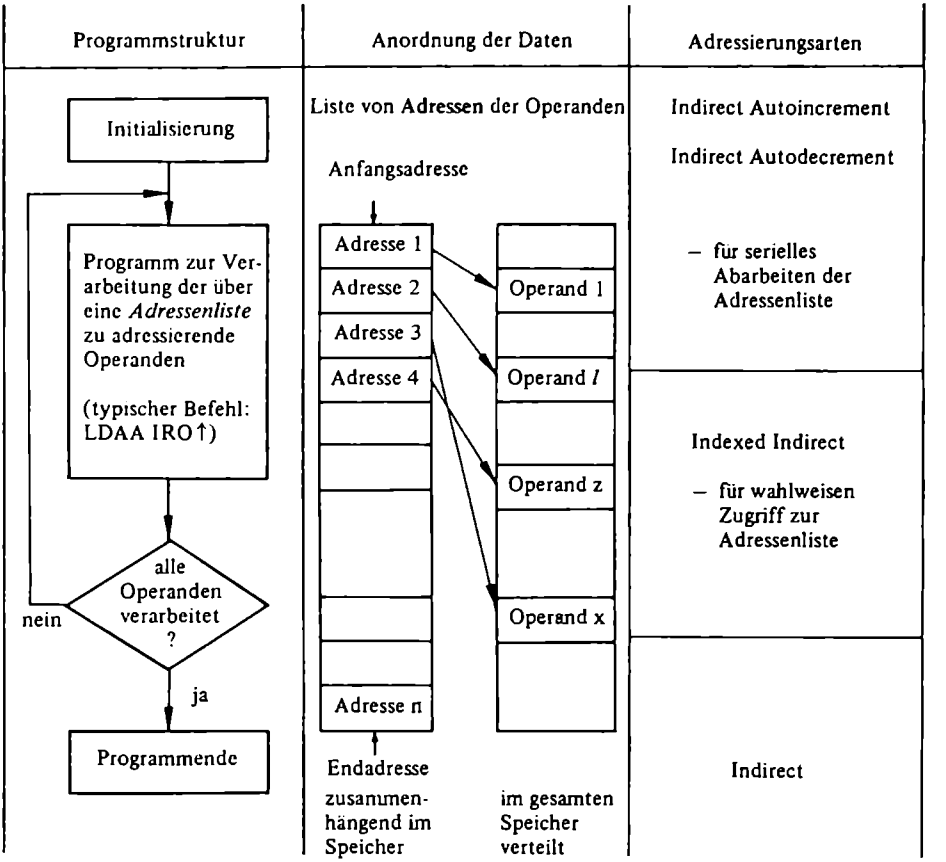
Flußdiagramm für eine Multiplikationsroutine

Multiplikationsroutine

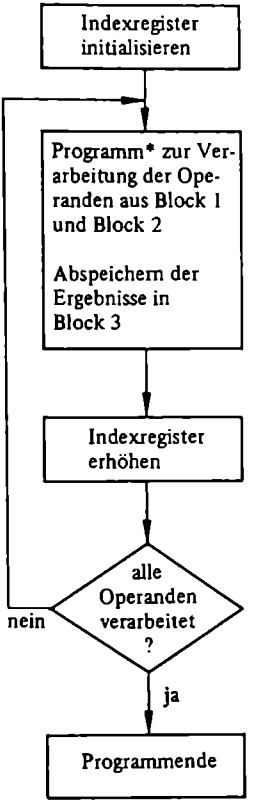
```
                *0                  /Startadresse
                LDAA #10
                STAA E  ZAHL10
                LDAB E  M2          /Multiplikator →ACCB
                CLRA                /0 →ACCA
MULT,           ASLA                /Arithmetic shift left ACCA
                                    /⟨B7⟩ →0; 0 →B0
                ROL E   PRODH       /Rotieren von PRODH nach links
                                    /⟨C⟩ →B0; ⟨B7⟩ →C
                ASLB                /Arithmetic shift left ACCB
                                    /0 →B0; ⟨B7⟩ →C
                BCC  NOCARY         /Carry = 0, dann Verzweigung
                                    /zu Nocary
                CLC                 /Carry löschen
                ADDA E  M1          /Akkumulator A + Multiplikand
                                    /⟨ACCA⟩+ ⟨M1⟩ →ACCA
                BCC  NOCARY         /Carry = 0, dann Verzweigung
                                    /zu Nocary
                INC E  PRODH        /bei der Addition ergab sich
                                    /ein Übertrag
                                    /⟨PRODH⟩+ 1 →PRODH
NOCARY,         DEC E  ZAHL10       /Zähler um 1 erniedrigen
                BGT  MULT           /Zähler > 0, zurückspringen
                                    /zu MULT
                STAA E  PRODL       /⟨ACCA⟩→ PRODL
ENDE,           BRA  ENDE

M1,             322
M2,             202
PRODL,          0
PRODH,          0
ZAHL10,         0
```

Beispiele zur Listenverarbeitung




Programmstruktur
Anordnung der Daten
Adressierungsarten
Initialisierung
Programm zur Verarbeitung der über eine *Adressenliste* zu adressierende Operanden
(typischer Befehl: LDAA IRO↑)
alle Operanden verarbeitet ?
nein
ja
Programmende
Liste von Adressen der Operanden
Anfangsadresse
Adresse 1
Adresse 2
Adresse 3
Adresse 4
Adresse n
Endadresse
zusammenhängend im Speicher
Operand 1
Operand *l*
Operand z
Operand x
im gesamten Speicher verteilt
Indirect Autoincrement
Indirect Autodecrement
– für serielles Abarbeiten der Adressenliste
Indexed Indirect
– für wahlweisen Zugriff zur Adressenliste
Indirect

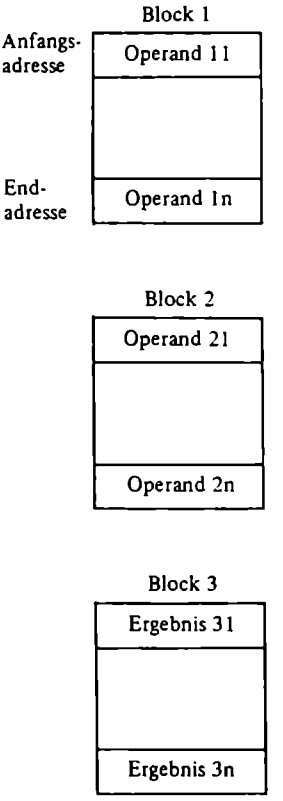
Programmstruktur	Anordnung der Daten	Adressierungsarten



*Programmbeispiel:

```
LDAA X  ADR1
LDAB X  ADR2
ABA
STAA X  ADR3
INX
```

Liste von Operanden und Ergebnissen



Indexed

– initialisieren eines Indexregisters auf einen Anfangswert, der nach jeder Abarbeitung eines Operanden erhöht wird

Kapitel 5
Der Mikroprozessor in Verbindung mit Peripheriegeräten

Ein Mikroprozessor wurde bisher vom Gesichtspunkt seines internen Aufbaus, seiner Baugruppen (Kapitel 2) und seines Befehlsvorrats (Kapitel 4) betrachtet. Dabei wurde die Betonung auf den Informations- und Datentransfer zwischen den Baugruppen Zentraleinheit und Speicher gelegt. Ein Mikrocomputer wäre in seiner Anwendbarkeit sehr stark eingeschränkt, wenn nur ein Informationsaustausch Zentraleinheit ↔ Speicher bestünde und nicht auch die Möglichkeit gegeben wäre, Information in Form von Daten und Steuersignalen mit der Peripherie, d.h. verschiedensten Eingabe- und Ausgabegeräten auszutauschen. Dabei ist an Geräte wie Teletype, Massenspeicher (Platte, Trommel, Band), Datensichtgeräte und D/A- bzw. A/D-Wandler gedacht, um nur einige zu nennen.

Das Ziel dieses Kapitels ist es, Möglichkeiten aufzuzeigen, wie ein Mikroprozessor mit Peripheriegeräten verbunden wird, wie der Datenaustausch vonstatten geht und welche Zusatzbausteine, Interface genannt, benötigt werden.

Folgende Datenaustauschverfahren werden behandelt und ihre jeweiligen Anwendungsgebiete vorgestellt:

- Programmierter Datenaustausch (programmed transfer)
- Datenaustausch durch Programmunterbrechung (programme interrupt)
- Datenaustausch durch direkten Speicherzugriff (direct memory access transfer = DMA-Transfer).

Ehe auf die einzelnen Datenaustauschverfahren eingegangen werden kann, müssen die verschiedenen Aufgaben und der prinzipielle Aufbau von Interface-Bausteinen erläutert werden. Da jeder Mikroprozessorhersteller etwas abgewandelte Bausteine anbietet, soll in diesem Kapitel nur so weit auf den Interface-Baustein eingegangen werden, wie es für das Verständnis der Eingabe- und Ausgabeverfahren notwendig ist.

5.1 Der Interface-Baustein

Mikrocomputer zeichnen sich häufig durch eine BUS-Struktur aus. Unter dem englischen Begriff „BUS" wird eine Gruppe paralleler Leitungen verstanden, die z.B. für den Datenaustausch zwischen Zentraleinheit und Speicher des Mikrocomputers zuständig ist. Derartige Gruppen sind z.B. die Adreßleitungen und die Datenleitungen, die deshalb oft auch als Adreßbus (addressbus) bzw. Datenbus (databus) bezeichnet werden.

Eine BUS-Struktur wird von vielen Mikroprozessorherstellern gewählt, um die Anzahl der Verbindungsleitungen niedrig zu halten. Der Mikroprozessor ist, wie jede andere integrierte Schaltung, auf einem Halbleiterchip untergebracht, an dem nur eine begrenzte Zahl von Anschlüssen vorgesehen ist, z.B. 40.

Die beiden häufigsten BUS-Strukturen sollen hier vorgestellt werden:

Einfach-BUS-Struktur

Daten und Adressen sowie Steuersignale werden auf demselben BUS im Zeitmultiplexbetrieb zwischen Zentraleinheit und Speicher sowie Peripherie ausgetauscht. Es ist somit nur *eine* Gruppe paralleler Leitungen vorhanden. Die Kennzeichnung der übertragenen Information, ob es sich z.B. um Adressen oder Daten handelt, muß über Kennungssignale, die der Mikroprozessor liefert, abgeleitet werden. Dazu werden zusätzliche Schaltkreise, z.B. Dekoder, verwendet.

Mehrfach-BUS-Struktur

Daten und Adressen sowie Steuersignale werden über getrennte BUS-Systeme übertragen. Der Datenbus ist dabei meistens als Zweirichtungsbus (bi-directional BUS) realisiert, d.h. Eingabe und Ausgabe von Daten zwischen Speicher sowie Peripherie erfolgt auf ein und demselben BUS, d.h. in beiden Richtungen (bi-directional) kann übertragen werden.

Der Adreßbus sorgt für die Übertragung der Adressen zum Speicher und zu den Ein-Ausgaberegistern.

Über die Steuerleitungen (Control-BUS) werden Signale zur Überwachung des Datentransfers innerhalb und außerhalb des Mikrocomputers übertragen.

Diese beschriebenen BUS-Strukturen bieten sich an, um eine Verbindung Mikroprozessor – Peripheriegerät aufzubauen.

Die Verbindungsleitungen umfassen:
Adreßleitungen, Datenleitungen und Steuerleitungen (Zeitsignale (z.B. Prozessortakt), Steuersignale (z.B. R/$\overline{W}$, interrupt request)).

Die Summe dieser Verbindungsleitungen wird oft auch als I/O-BUS bezeichnet. Durch diesen I/O-BUS sind alle Peripheriegeräte über *ihr* Interface mit dem Mikroprozessor (CPU) verbunden (Bild 5.1).

5.1.1 Anforderungen an ein Interface

Die Anforderungen, die an einen Interface-Baustein gestellt werden, lassen sich aus Bild 5.1 ableiten. Sie ergeben sich, sobald der Mikroprozessor mit einem Peripheriegerät verbunden werden soll. Ebenso wie für den Mikroprozessor müssen für das Peripheriegerät bestimmte Bedingungen erfüllt sein, z.B. bezüglich Technologie, Datenformat und Zeitsignalen. Eine Aufgabe des Interface besteht darin, den Mikroprozessor und das Peripheriegerät aneinander anzupassen. Diese Anpassung kann z.B. in der Weise geschehen, daß die für den Mikroprozessor gültigen TTL-Pegel so verändert werden, daß ein Drucker angesteuert werden kann. Im Fall des Datenformats ist es oft notwendig, die im Mikroprozessor parallel verarbeiteten Daten in ein serielles Format für ein Ausgabegerät, z.B. Teletype, umzuwandeln. Ein wichtiges und oft

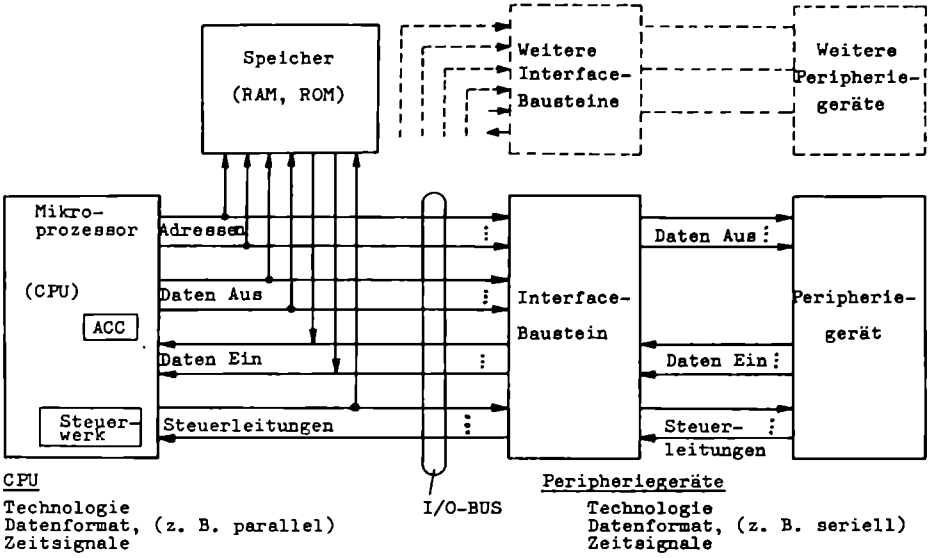


Bild 5.1: Verbindungen zwischen Mikroprozessor und Peripheriegerät

sehr schwer zu lösendes Problem besteht in der Anpassung der für den Mikroprozessor gültigen Zeitsignale an die Zeitbedingungen des Peripheriegerätes. Ein Teil dieses Kapitels, der sich mit den Datenaustauschverfahren befaßt, zeigt Möglichkeiten auf, um dieses Problem zu lösen.

Im folgenden sind in Tabellenform Anforderungen zusammengestellt, denen ein Interface-Baustein gerecht werden sollte:

- Informationsaustausch zwischen Peripheriegerät und Mikroprozessor soll in beiden Richtungen erfolgen können. Dabei müssen unterschiedliche Geschwindigkeiten zwischen Mikroprozessor und Peripheriegerät ausgeglichen werden (→ asynchroner Transfer, Sense-Leitung, Abfragen, Device-Flag).
- Ausgleich von verschiedenen Architekturen zwischen Mikroprozessor und Peripheriegerät (→ Umwandlung von Datenformaten, Pegelanpassung).
- Befehlsdekodierung für Peripheriegeräte (→ Einstellen eines Betriebszustandes im Peripheriegerät, z.B. bei einem Bandgerät: Motor auf Linkslauf stellen).
- Statusinformation über Peripheriegeräte dem Mikroprozessor mitteilen.
- Auswahl eines bestimmten Peripheriegerätes über eine Adressendekodierlogik.
- Weiterleitung einer Bedienungsanforderung des Peripheriegerätes an den Mikroprozessor (→ Programmunterbrechung = programme interrupt).
- Das Interface muß vielseitig einsetzbar sein, um möglichst vielen Anforderungen von Peripheriegeräten gerecht zu werden. Es sollte deshalb anwendungsspezifisch programmierbar sein, um verschiedene Peripheriegeräte an den Mikroprozessor anpassen zu können. Über ein Programm, das die charakteristischen Eigenschaften des anzupassenden Peripheriegerätes enthält bezüglich Datenformat und Zeitsignalen, wird der Interface-Baustein für die neue Aufgabe befähigt.

5.1.2 Aufbau verschiedener Interface-Bausteine

In Bild 5.2 und 5.3 sind zwei typische Interface-Bausteine dargestellt, wie sie in Mikrocomputern eingesetzt werden.

Das in Bild 5.2 dargestellte Interface besteht aus Zwischenspeichern für die Ausgabe und Mulitplexern für die Eingabe. Der Transfer zwischen der Peripherie erfolgt über die Datenleitungen. Die Art des Transfers (Eingabe/Ausgabe) und die Auswahl des gewünschten Peripheriegerätes werden über die Steuerlogik und die Adressendekodierlogik durchgeführt. Die beiden mit „Eingabe" bzw. „Ausgabe" bezeichneten Signale werden, wie später noch gezeigt wird, für den asynchronen Transfer benötigt.

Der Anschluß mehrerer Interface-Bausteine auf dem Datenbus setzt voraus, daß es sich bei den Ausgängen um „open collector"-Ausgänge oder um „tri-state"-Ausgänge handelt. Diese stellen sicher, daß erstens ein Zweirichtungstransfer möglich ist, und zweitens die Belastung des Datenbus durch Interface-Bausteine klein gehalten werden kann, die nicht an einem Datentransfer beteiligt sind, denn ein „tri-state"-Ausgang kann für diesen Fall in einen hochohmigen Zustand gebracht werden. Dadurch wird der Datenbus nur geringfügig belastet.

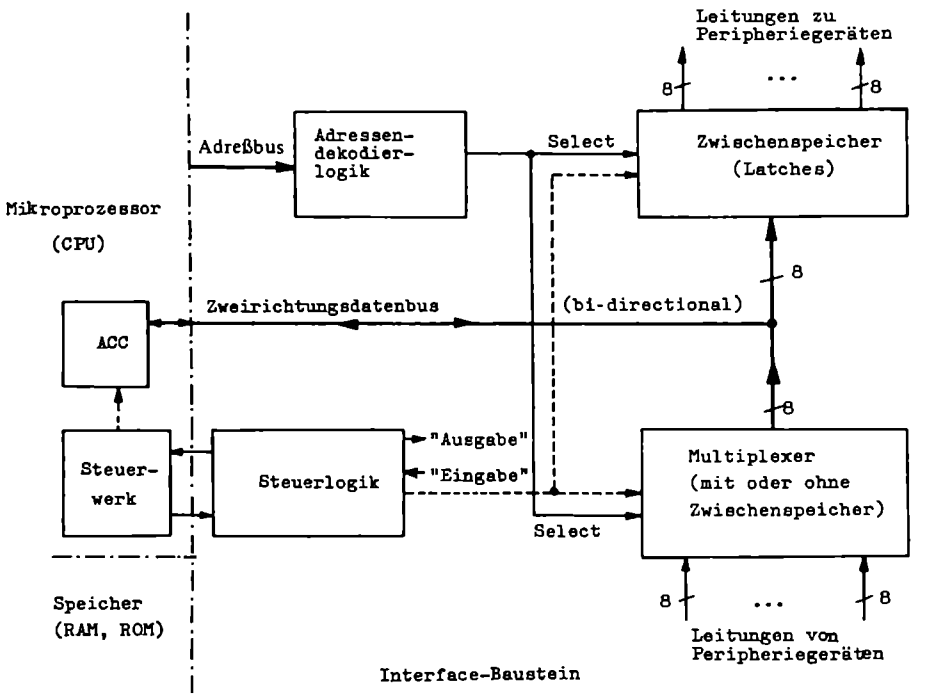


Bild 5.2: Interface-Baustein auf Zweirichtungsbus (bi-directional BUS) (getrennte Ein-/Ausgabebausteine)

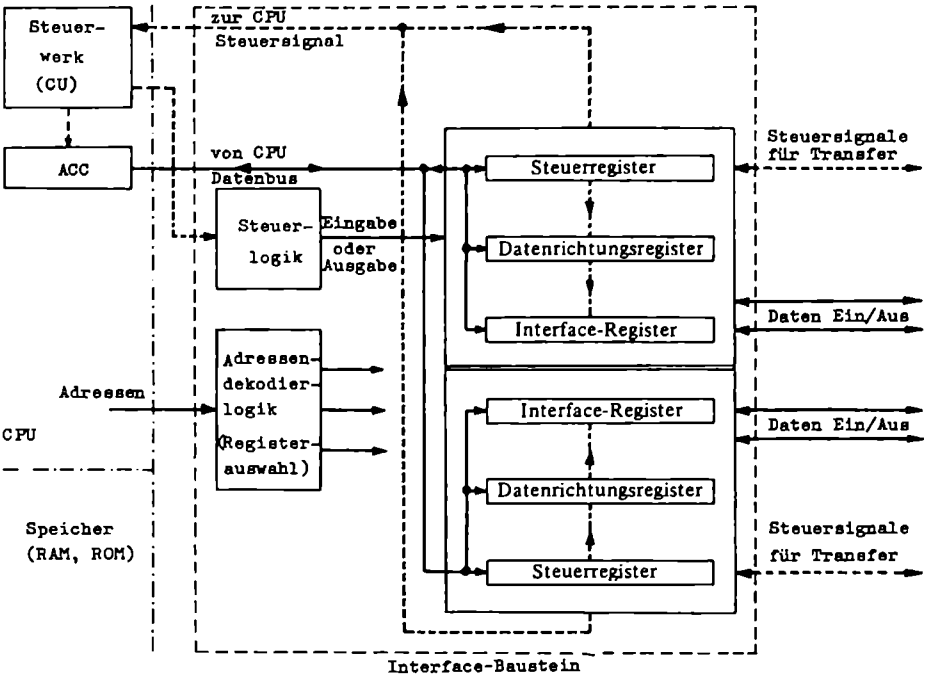


Bild 5.3: Programmierbares Interface

Das in Bild 5.2 dargestellte Interface findet Anwendung, wenn ein asynchroner Transfer vorliegt und deshalb die Information zwischengespeichert werden muß, bevor sie vom oder zum Peripheriegerät übertragen wird.

Das in Bild 5.3 gezeigte Interface stellt eine Weiterentwicklung und Verbesserung des in Bild 5.2 behandelten dar. Dieses Interface kommt den in Abschnitt 5.1.1 gestellten Anforderungen sehr nahe. Die Hauptmerkmale dieses Bausteins sind:

- Drei per Programm (software) adressierbare Register, die die Funktion des Bausteins festlegen.
  1. Datenrichtungsregister legt fest, ob eine Leitung des Interface-Registers als Eingang oder Ausgang betrieben wird.
  2. Steuerregister legt die Funktion der das Peripheriegerät steuernden Statussignale fest (→ Datentransfersteuerung: Mikroprozessor ⟷ Peripheriegerät).
  3. Interface-Register ermöglicht den Transfer der über die Datenleitungen übertragenen Information vom und zum Peripheriegerät.
- Auswahl des Peripheriegerätes über die Adressendekodierlogik des Interface-Bausteins.
- Verbindungsleitungen, die vom Steuerregister gesteuert werden und dem Mikroprozessor mitteilen, wann immer ein Peripheriegerät bedient werden soll (interrupt request). → Status des Peripheriegerätes wird dem Mikroprozessor mitgeteilt (Abfrage des Steuerregisters mit Testbefehlen; siehe Kapitel 4, Tabelle 4.4).

Mit diesem Interface läßt sich eine Anpassung des Mikroprozessors an Peripheriegeräte erreichen, die bis zu 16 parallele Ein-/Ausgabeleitungen besitzen. Der augenblickliche Zustand des Peripheriegerätes läßt sich über das Steuerregister und die damit verbundenen Steuerleitungen erfassen und gegebenenfalls abändern.

Das zuletzt besprochene Interface (Bild 5.3) macht deutlich, wie wichtig bei einem Mikroprozessor Testbefehle sein können, die sich auch auf den Inhalt von Speicherplätzen anwenden lassen (→ Erfassen des Status des Peripheriegerätes), denn die Register innerhalb des Interface sind für den Mikroprozessor nichts anderes als Speicherplätze, die über Adreßleitungen angesprochen werden können, die ausgelesen werden können und in die eingeschrieben werden kann.

Die Eingabe- bzw. Ausgabebefehle werden damit durch die Befehle „Load" und „Store" dargestellt (s. Kapitel 4, Tabelle 4.5).

5.2 Der Datenaustausch zwischen Mikroprozessor und Peripheriegerät

Der Datenaustausch umfaßt die Eingabe und Ausgabe von Information (Daten, Steuersignale) zwischen dem Mikrocomputer und den Peripheriegeräten.

Wie schon kurz angedeutet, kann dieser Austausch von Information auf verschiedene Arten (→ Datenaustauschverfahren, unterschiedliche Ein-/Ausgabebefehle) vorgenommen werden.

Erstens:

Es sind Eingabe- und Ausgabebefehle mit dem Mnemonic

IN für die Eingabe

und

OUT für die Ausgabe vorhanden.

Zusätzlich zum Mnemonic muß noch eine Adresse festgelegt werden, die das Interface auswählt, über das der Datenaustausch mit dem Peripheriegerät vorgenommen wird.

Bei Mikroprozessoren dient häufig ein Akkumulator als Empfangs- (bei Befehl IN) und als Senderegister (bei Befehl OUT) für die auszutauschende Information.

Zweitens:

Es stehen keine besonderen Ein-/Ausgabebefehle zur Verfügung, sondern der Datenaustausch zwischen Mikroprozessor und Peripheriegerät erfolgt unter Zuhilfenahme eines Interface-Bausteins in derselben Weise, wie zwischen Mikroprozessor und Speicher. Für den Mikroprozessor sind somit der Speicher und die Register des Interface-Bausteins nichts anderes als Speicherplätze. Die Ein-/Ausgabe wird mit den Befehlen

LOAD für die Eingabe

und

STORE für die Ausgabe durchgeführt.

Die zusätzlich bei „Load" und „Store" angegebene Adresse ist die Adresse des Interface-Registers, über das der Datenaustausch zwischen Mikroprozessor und Peripheriegerät durchgeführt wird.

Ein weiterer Vorteil dieser Art der Ein-/Ausgabe besteht darin, daß Testbefehle, die auf den Inhalt von Speicherplätzen angewendet werden, in gleicher Weise auch mit dem Registerinhalt des Interface-Bausteins arbeiten können.

Der Datenaustausch kann auf verschiedene Weise vom Mikroprozessor und dem Interface-Baustein oder vom Interface-Baustein allein durchgeführt werden. Welche Verfahren im einzelnen verwendet werden, soll nun gezeigt werden.

5.2.1 Programmierter Datenaustausch (programmed transfer)

Beim programmierten Datenaustausch unterscheidet man zwei Arten:
- unbedingter Austausch,
- bedingter Austausch.

Es wird unterschieden, ob der Austausch von einer externen Bedingung, z.B. „Drukker zur Ausgabe eines Buchstabens bereit", abhängig ist oder nicht.

Unbedingter Austausch (unconditional transfer)

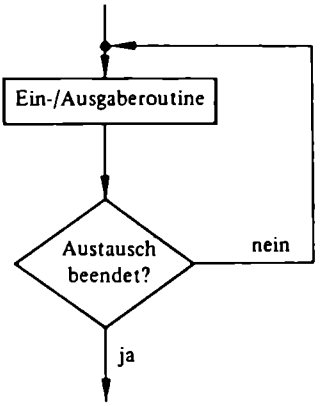


Bild 5.4: Unbedingter Austausch

Diese Art des Datenaustausches kommt in der Praxis selten vor, da bei diesem Transfer der Status des angesprochenen Peripheriegerätes, ob dasselbe „fertig" oder „beschäftigt" ist, nicht nachgeprüft wird. Das Peripheriegerät muß in der Zeit Information ausgeben oder bereitstellen, die vergeht, bis der Mikroprozessor wieder zu einer Eingabe bzw. Ausgabe bereit ist. Ist dies nicht der Fall, so führt dies zu einem Informationsverlust. Mit anderen Worten: Das Peripheriegerät ist den vom Mikroprozessor ausgesendeten Steuer- und Zeitsignalen unterworfen. Es handelt sich hier um einen vom Mikroprozessor synchronisierten Transfer.

Dieser Transfer kann immer nur dann angewandt werden, wenn von einem Peripheriegerät die Zeitbedingungen für die Ein-/Ausgabe zeitlich konstant und bekannt sind.

Bedingter Austausch (conditional transfer)

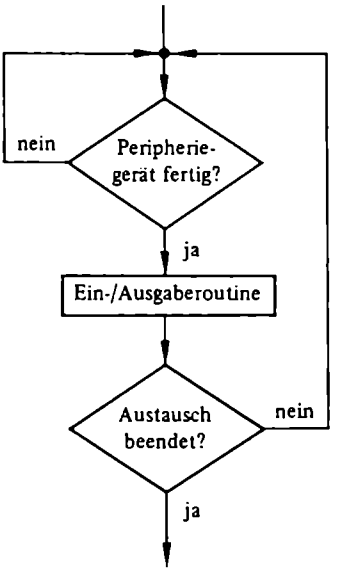


Bild 5.5: Bedingter Austausch

Der bedingte Austausch findet bei Mikroprozessoren breite Anwendung. Die Ein-/Ausgaberoutine (Bild 5.4) ist durch eine Abfrageroutine (Peripheriegerät fertig?) erweitert worden (Bild 5.5). Die Durchführung des bedingten Austausches läuft folgendermaßen ab:

- Adressierung des zum Peripheriegerät gehörenden Interface-Bausteins.
- Abfragen eines Steuerregisters (Statusregister[1])) des Interface-Bausteins, ob das Peripheriegerät bedient werden soll. Zu diesem Zweck wird eine bestimmte Leitung („Sense"-Leitung oder auch „Device-Flag") einem Bit des Steuerregisters zugeordnet. Im Falle einer Eingabe wird durch Setzen (Bit = 1) des bestimmten Bits mitgeteilt: Peripheriegerät ist zur Eingabe bereit. Für eine Ausgabe gilt sinngemäß: Peripheriegerät ist zur Aufnahme von Information bereit.
- Ist die Abfrage positiv, so kann der Mikroprozessor seine Ein-/Ausgaberoutine abarbeiten.
- Das Ende des Transfers bildet eine Abfrage, ob der Austausch beendet ist oder nicht. Ist er nicht beendet, so erfolgt ein Rücksprung, und ein abermaliges Testen des Steuerregisters ist die Folge. Ist der Austausch jedoch beendet, so kehrt der Mikroprozessor zum Hauptprogramm zurück.

[1]) Im Statusregister eines Interface-Bausteins können auch die Sense-Leitungen aller anderen Peripheriegeräte zusammengefaßt werden. Auf diese Weise läßt sich durch Adressierung eines Interface-Bausteins eine der Registerbreite entsprechende Anzahl von Peripheriegeräten auf ihren „Fertig"-Zustand hin überprüfen.

Der Vorteil gegenüber dem unbedingten Austausch besteht darin, daß eine Synchronisation zwischen dem Mikroprozessor und dem Peripheriegerät ermöglicht wird.

Durch diese Synchronisation kann es jedoch dazu kommen, daß der Mikroprozessor auf das „Fertig"-Signal eines langsamen Peripheriegerätes lange warten muß und so Zeit vergeudet wird. Dieser Nachteil kann auf zweierlei Weise umgangen werden:

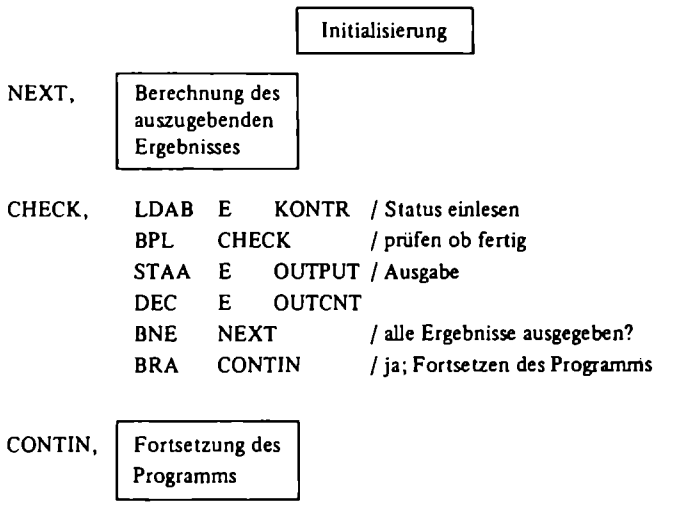
- Die Abfrage des „Fertig"-Signals erfolgt in einem bestimmten Zeitraster, d.h. die Abfrageschleife wird durch eine einmalige Abfrage des Zustandes des Peripheriegerätes ersetzt, die nach einer bestimmten Zeit wiederholt wird.
- Ersatz des bedingten Austausches (conditional transfer) durch den Datenaustausch durch Programmunterbrechung (programme interrupt).

Programmierter Datenaustausch (Programmbeispiel)

Vereinbarungen zum Programm

- Mikroprozessor gibt Ergebnisse an ein Ausgabegerät aus.
- Die Abfrage des Peripheriegerätes erfolgt durch Einlesen eines Kontrollregisters des Interface-Bausteins.
- Der Zustand des Peripheriegerätes ist aus Bit 7 des Kontrollregisters zu entnehmen.
- Das Peripheriegerät ist zur Ausgabe bereit, wenn Bit 7 logisch „1" ist.
- Das Ergebnis, das im Akkumulator A steht, wird an das Peripheriegerät ausgegeben, indem die Speicherstelle (Register) namens OUTPUT geladen wird.
- Die Label NEXT, CHECK und CONTIN sind symbolische Adressen.

Programmbeispiel



Eine weitere Möglichkeit des bedingten Austausches (conditional transfer) steht mit dem sogenannten Handshake-Transfer, einem asynchronen Transfer, zur Verfügung. Er wird angewandt, sobald eine genaue Kontrolle sowie erhöhte Sicherheit bei einem Datenaustausch gefordert sind, z.B. beim Datenaustausch eines Mikroprozessors und

eines Datenterminals. Der Handshake-Transfer ist gekennzeichnet durch eine Art „Zwiegespräch" zwischen dem Mikroprozessor und seinem Peripheriegerät. Das wesentliche bei einem Handshake-Transfer besteht darin, daß der Transfer von Signalen überwacht wird, die sowohl vom Mikroprozessor als auch vom Peripheriegerät ausgesendet bzw. abgefragt werden (→ „Zwiegespräch").

In Bild 5.6 sind ein Impulsdiagramm und das zugehörige Flußdiagramm eines Handshake-Transfers dargestellt.

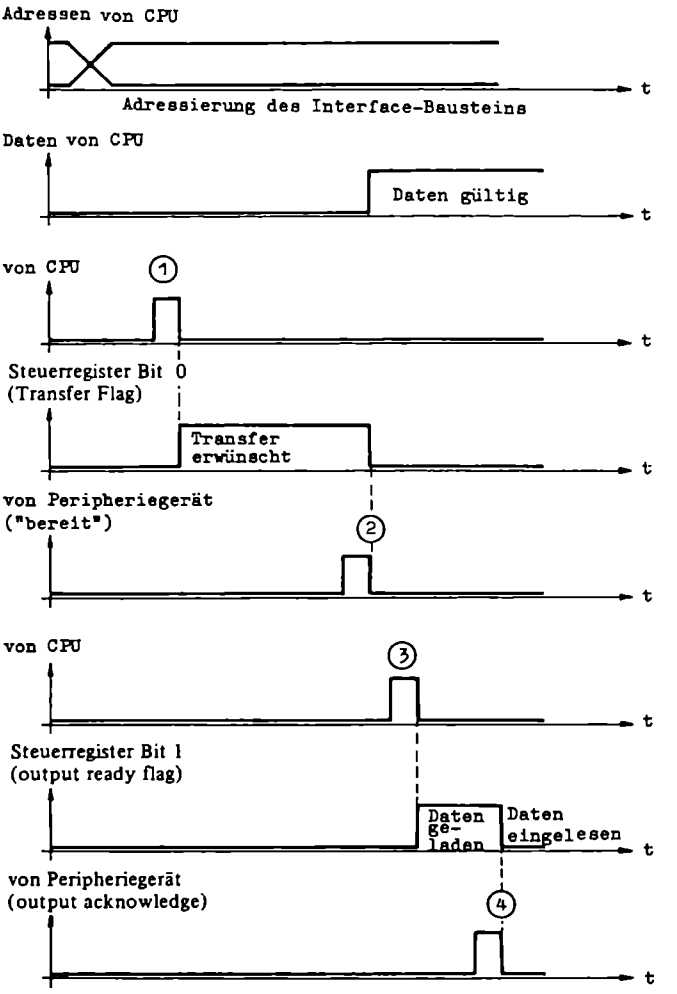


Bild 5.6a: Impulsdiagramm eines Handshake-Transfers (Ausgabe)

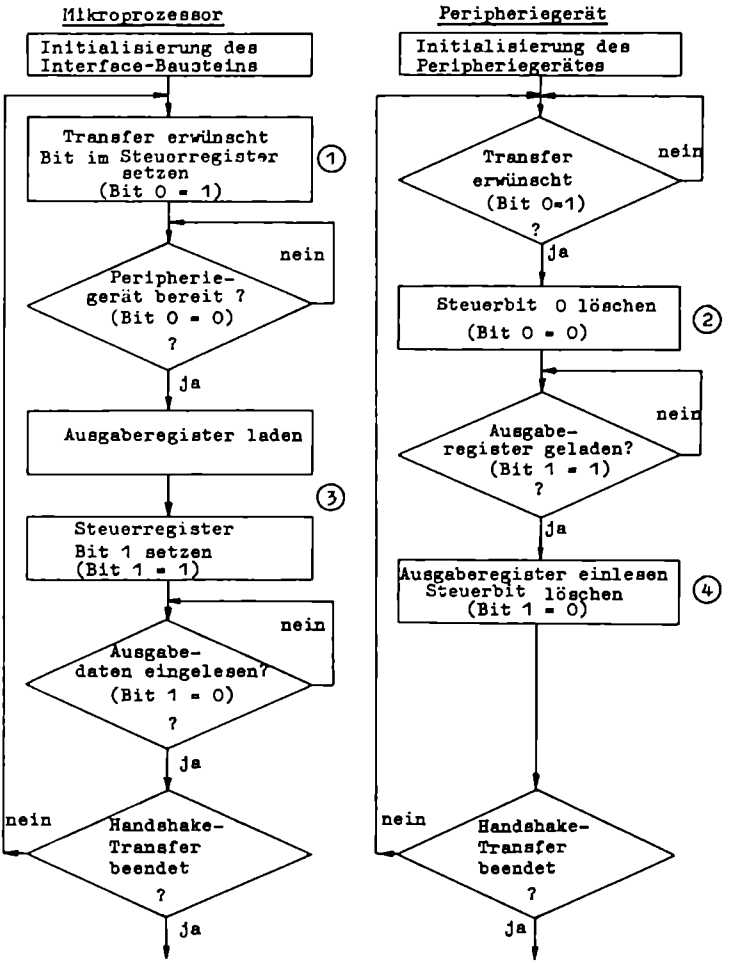


Bild 5.6b: Flußdiagramm zum Handshake-Transfer

Typische Signale während eines solchen Transfers sind:

	von CPU	von Peripherie
	Transfer erwünscht (transfer request)	Transfer gewährt (transfer grant)
Für Ausgabe:	Daten sind geladen (output ready)	Daten eingelesen (output acknowledge)
Für Eingabe:	Daten eingelesen (input acknowledge)	Daten bereit (input ready)

Die Anzahl der gegenseitigen Abfragen bleibt dem Entwickler überlassen, es gibt dafür keine feste Regel, jedoch kann mit zwei oder vier Abfragen schon eine sichere Übertragung gewährleistet werden.

Erläuterung eines Handshake-Transfers (Bild 5.6 und 5.7)

Am Beispiel einer Ausgabe wird der Datenaustausch mittels Handshake erläutert. Es werden die folgenden Phasen durchlaufen (siehe Bild 5.6):

(1) Der Mikroprozessor adressiert den zum Peripheriegerät gehörenden Interface-Baustein und prüft den Zustand des Peripheriegerätes.

(2) Laden des Interface-Registers mit den auszugebenden Daten, wenn das Peripheriegerät fertig ist.

(3) Über eine Steuerleitung des Interface-Bausteins wird dem Peripheriegerät mitgeteilt, daß Daten zur Ausgabe bereitstehen.

(4) Das Peripheriegerät liest die im Interface-Baustein zwischengespeicherten Daten ein und teilt dem Mikroprozessor den Abschluß des Einlesens durch ein Steuersignal „Ausgabedaten eingelesen" an das Interface mit (output acknowledge).
Der Mikroprozessor lädt das Interface-Register wiederum mit neuen Daten, wenn obiges Steuersignal erkannt wurde und der Datenaustausch noch nicht beendet ist.

Den einzelnen Phasen werden die folgenden Signale des Handshake-Transfers zugeordnet (siehe Bild 5.7):

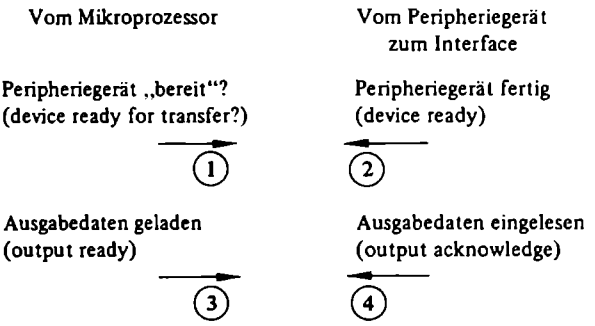


Bild 5.7: Handshake-Transfer

Allen in 5.2.1 behandelten Datenaustauschverfahren (unbedingter -, bedingter -, Handshake-Transfer) ist eines gemeinsam: Sie werden unter Kontrolle eines Mikroprozessorprogramms durchgeführt.

Der „programme interrupt"- und der DMA-Transfer, die jetzt behandelt werden, unterliegen außerdem noch der Kontrolle des Interface-Bausteins oder wie im Falle des DMA-Transfers, ausschließlich der des Interface-Bausteins.

5.2.2 Datenaustausch durch Programmunterbrechung (programme interrupt)

Beim „programme interrupt" erfolgt der Informationsaustausch unter Programmkontrolle, den Zeitpunkt, zu dem der Transfer ausgelöst wird, bestimmt jedoch ein Peripheriegerät. Dieser Transfer wird somit vom Mikroprozessor kontrolliert, aber vom Peripheriegerät initiiert. Dieser Sachverhalt ist stark vereinfacht in Bild 5.8 verdeutlicht.

Der Mikroprozessor bearbeitet einen Programmabschnitt, den er bei Erhalt eines Interrupts verläßt und sodann mit der Abarbeitung einer Ein-/Ausgaberoutine beginnt, die die Bedienung des den Interrupt verursachenden Peripheriegerätes sicherstellt. Nach Abschluß dieser Ein-/Ausgaberoutine erfolgt die Fortsetzung des Programms dort, wo der Interrupt erfolgte (vgl. Anrufbefehle für Unterprogramme, Abschnitt 4.9).

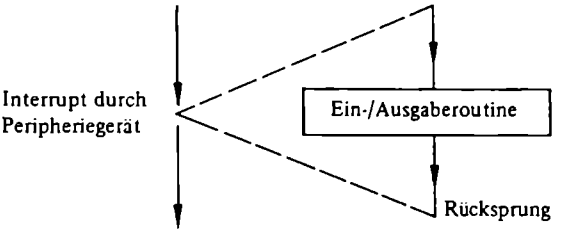


Bild 5.8: Programmunterbrechung

Im Zusammenhang mit Interrupts müssen noch einige Punkte näher betrachtet werden; dies soll an ausgewählten Interruptsystemen verdeutlicht werden. Es werden die von der Software auszuführenden Aufgaben bei der Behandlung von Interrupts besprochen und zuletzt wird gezeigt, welche Besonderheiten beim Anwenden des „programme interrupt" zu beachten sind. Da jeder Mikroprozessor „Interrupts" auf leicht abgewandelte Weise bearbeitet, wird in der weiteren Betrachtung auf eine grundlegende Darstellung Wert gelegt.

5.2.2.1 Abgrenzung gegen Systeme ohne Interrupt

Ein Mikroprozessor führt unter der Kontrolle eines Programms eine fest vorgegebene Befehlsfolge aus. Auf die Anforderung eines Peripheriegerätes, d.h. auf ein außerhalb des Mikroprozessorsystems stattfindendes Ereignis, kann der Mikroprozessor nur eingehen, wenn der Programmierer eine entsprechende Abfrage einer „Sense"-Leitung (→ conditional transfer) vorgesehen hat. Das Problem einer solchen Abfrage besteht darin, daß externe Ereignisse gewöhnlich asynchron sind, d.h. in keinem definierten Zeitverhältnis zu den im Mikroprozessor ablaufenden Vorgängen stehen. Bei einer Abfrage kann es deshalb durchaus geschehen, daß kurz nach einer Abfrage der „Sense"-Leitung, die ergab, daß keine Aktivierung derselben stattgefunden hatte, ein Peripheriegerät bedient werden möchte. Diese nachträgliche Aktivierung der „Sense"-Leitung wird erst erkannt, wenn der Mikroprozessor abermals eine Abfrage derselben vornimmt. Es verstreicht somit eine gewisse Zeit, bis der Mikroprozessor die Anforderung der Peripherie erkennt. Da eine Abfrage immer mit einem gewissen Zeitauf-

wand verbunden ist, muß sich der Entwickler genau überlegen, ob er sich den aus der wiederholten Abfrage der „Sense"-Leitung ergebenden zusätzlichen Zeitbedarf für die Programmverarbeitung leisten kann oder nicht. Um dieses Problem zu umgehen, bedient man sich des Interrupts. Der Mikroprozessor bearbeitet sein Programm und bedient die Peripherie nur, wenn diese den Mikroprozessor um eine Bedienung nachsucht. Somit entfällt die Abfrage einer „Sense"-Leitung.

5.2.2.2 Interruptsysteme und ihre Bedienung durch den Mikroprozessor

In Bild 5.9 sind verschiedene Interruptsysteme dargestellt, deren Behandlung durch den Mikroprozessor in diesem Abschnitt erläutert wird. Anhand eines Flußdiagramms (Bild 5.10) werden die einzelnen Phasen von der Erkennung eines Interrupts über die Bedienung bis zur Rückkehr zum Hauptprogramm besprochen.

Bedienung eines Interrupts

1. Die Peripherie löst einen Interrupt aus (interrupt request).
2. Der Mikroprozessor erkennt den Interrupt zu einem im Steuerwerk festgelegten Zeitpunkt meistens am Ende des gerade ausgeführten Befehls.
3. Einige Mikroprozessoren teilen über einen separaten Anschluß mit, daß der Interrupt erkannt wurde (interrupt acknowledge).
4. Setzen eines internen Bit, um weitere Interrupts zu verhindern (Interrupt-Mask-Bit).
5. Per Programm muß dafür gesorgt werden, daß der augenblickliche Zustand des Mikroprozessors abgespeichert wird (Programmzähler (PC), Registerinhalte, Statusflags). Der Stack wird vorteilhaft als Speicherplatz benutzt. Viele Mikroprozessoren führen diese Operationen in eigener Regie durch. Dies kann den einen Nachteil haben, daß auch Register abgespeichert werden, deren Inhalte durch die Interruptroutine nicht zerstört werden.
6. Ablauf einer mittels Software oder Hardware verwirklichten Identifikation des Interrupts um festzustellen, welches Peripheriegerät den Interrupt ausgelöst hatte.
   Die Art der Identifikation ist maßgebend für die Geschwindigkeit, mit der der Mikroprozessor auf einen Interrupt reagiert, d.h. auf das externe Ereignis anspricht.
   Zur Identifikation bedient man sich verschiedener Software- und Hardware-Methoden:

   Software-Realisierung (polling program)

   Dieses Programm besteht aus einer „Kettenschaltung" von Steuerregister- oder „Sense"-Leitungsabfragen[1]). Die Identifikation ist durch eine eindeutige Zuordnung von Steuerregisteradresse und Peripheriegerät möglich. Der Vorteil dieses Programms ist darin zu sehen, daß durch die „Kettenschaltung" den einzelnen an das Interruptsystem angeschlossenen Peripheriegeräten per Programm (Software) eine Priorität zugeordnet wird. Es wird somit erreicht, daß der Interrupt mit der höchsten Priorität zuerst bedient wird (siehe Beispiel: Polling Routine).

[1]) Dieses Programm kann auch als „Check-Chain" bezeichnet werden.

Hardware-Realisierung

Der Nachteil, daß dieses Abfragen sehr zeitintensiv ist, kann dadurch umgangen werden, daß alle „Sense"-Leitungen zu einem Flag- oder Interruptstatuswort zusammengefaßt werden, indem sie gemeinsam an ein Interface-Register angeschlossen werden. Das jeweils eingelesene Wort kann als Zeigeradresse zu einem Interrupt-Bedienungsprogramm verwendet werden (vectored interrupt). Die Priorität muß jedoch durch Hardware realisiert werden (Prioritätsschaltkreise; verschiedene Interrupteingänge am Mikroprozessor; Bild 5.9).

7. Nach der Identifikation des Interrupts erfolgt die Bedienung des Peripheriegerätes oder allgemeiner das Abarbeiten des Interrupt-Bedienungsprogramms (interrupt service routine). Bestandteil dieses Programms kann das Rücksetzen des internen „Interrupt-Mask-Bit" sein, um weitere Interrupts zuzulassen, d.h. solche höherer Priorität (→ interrupt nesting). Als Befehle stehen EI (enable interrupt) zum Rücksetzen des Mask-Bit bzw. DI (disable interrupt) zum Setzen des Mask-Bit zur Verfügung.
8. Am Ende der Interrupt-Bedienungsroutine müssen die zu Beginn abgespeicherten Registerinhalte wieder zurückgeschrieben werden. Wenn vorhanden, so bedient man sich des Befehls RTI (return from interrupt), der den Mikroprozessor wieder in den Zustand vor dem Auftreten des Interrupts bringt.
9. Der Mikroprozessor setzt das Programm dort weiter fort, wo er durch die Interruptanforderung (interrupt request) unterbrochen wurde.

Interruptsysteme

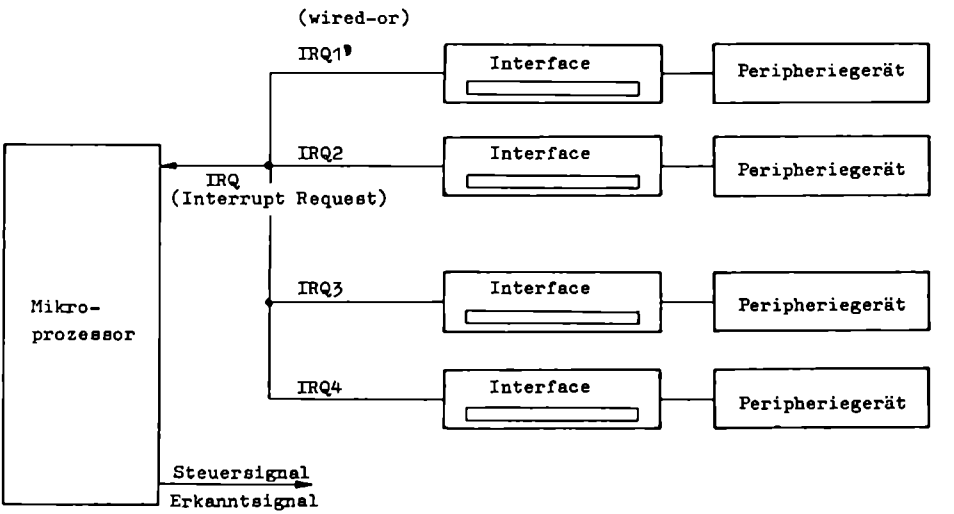


Bild 5.9a: Interruptsystem mit einer gemeinsamen Interruptleitung (wired-or). Identifikation über Polling Routine

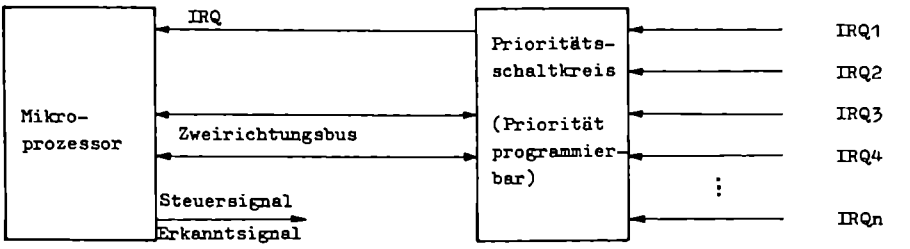


Bild 5.9b: Interruptsystem mit Prioritätsschaltkreis. Identifikation über Zeigeradresse

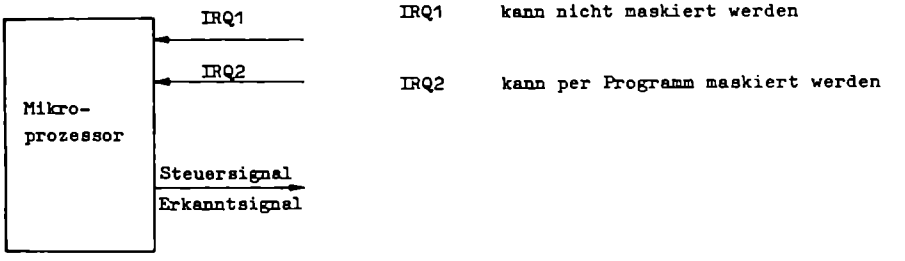


Bild 5.9c: Interruptsystem mit festgelegter Priorität (IRQ1 hat höhere Priorität als IRQ2)

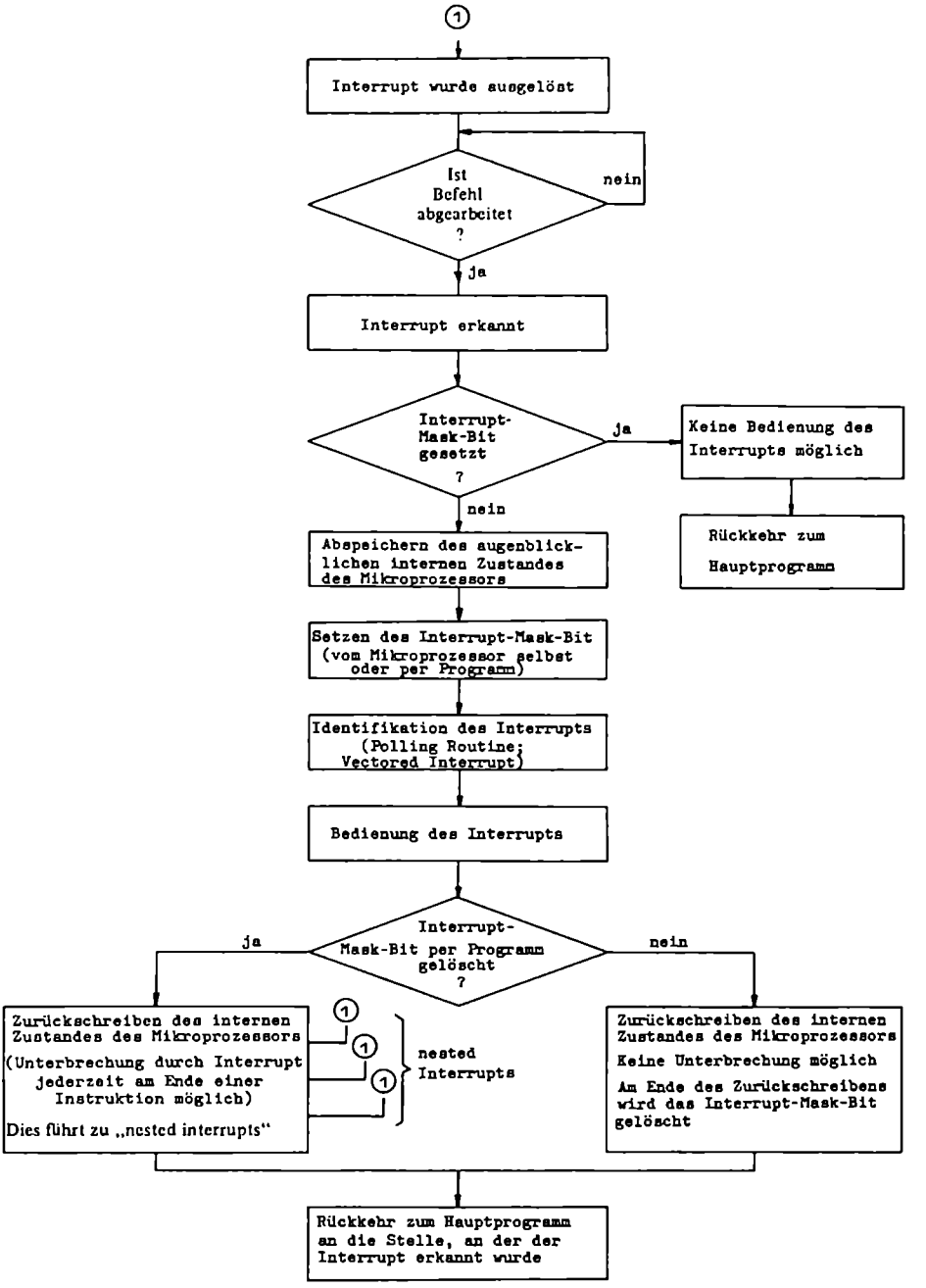
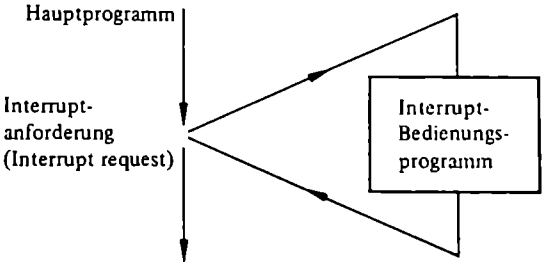


Bild 5.10: Bedienung eines Interrupts

Programmbeispiel und Flußdiagramm

A) Polling Routine

Vereinbarung zum Programm
- Ausgabe von Ergebnissen an die Peripherie
- Es seien 3 gleiche Peripheriegeräte angeschlossen
- Identifikation des unterbrechenden Gerätes über *Polling Routine*



- Kann der Interrupt nicht identifiziert werden, so erfolgt die Ausgabe einer Fehlermeldung

Polling Routine

```
CHECK1,  INX                    } /Indexregister erhöhen
         LDAB E KONTR1          }
         BPL  CHECK2            }
         BRA  RESOUT            }
CHECK2,  INX                    }
         LDAB E KONTR2          }
         BPL  CHECK3            }
         BRA  RESOUT            } Identifikation
CHECK3,  INX                    }
         LDAB E KONTR3          }
         BPL  ERROR             }
         BRA  RESOUT            }
ERROR,   Identifikation         }
         des Interrupts         }
         nicht möglich          }
         Ausgabe einer          }
         Fehlermeldung          }
RESOUT,  LDAA E RESULT          } Bedienungsroutine
         STAA X OUTPUT          }
         RTI                    }
```

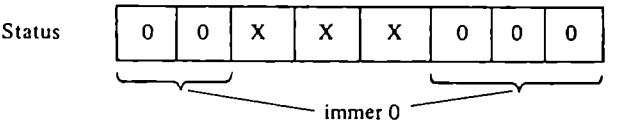
B) Vectored Interrupt

Vereinbarungen zum Programm

- Bedienung von 8 Peripheriegeräten
- Identifikation des unterbrechenden Gerätes über „vectored interrupt"
- Die 8 Statusleitungen werden auf eine Prioritätsschaltung geführt, deren 3 Ausgänge den entsprechenden Binär-Code-Wert des durchgelassenen Interrupts besitzen.

  z.B. Interrupt 7 → 111_2

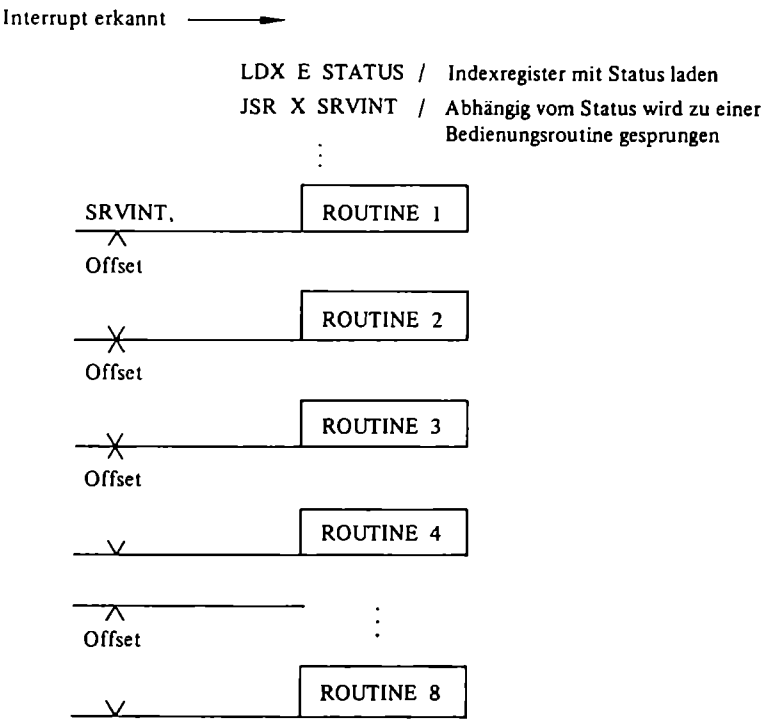
  Interrupt 4 → 100_2



- Abhängig von diesem Codewort ergibt sich eine Adresse zur Bedienungsroutine

  XXX = Binär-Code-Wert

Vectored interrupt



5.2.2.3 Abschließende Betrachtungen zur Programmunterbrechung (programme interrupt)

Das Polling Programm hat gezeigt, daß auch eine Programmunterbrechung viel Zeit benötigen kann, bis der Interrupt erkannt ist und mit dem Bedienungsprogramm begonnen werden kann. Es wird insbesondere Zeit benötigt für:

- Abarbeiten des laufenden Befehls,
- Abspeichern der Registerinhalte,
- Identifikation des Interrupts (nicht bei vectored interrupt),
- Abarbeitung des Bedienungsprogramms.

Zusammengefaßt: Entscheidend sind die Arbeitsgeschwindigkeit des Mikroprozessors sowie die Leistungsfähigkeit seiner Ein-/Ausgabebefehle und die Schnelligkeit der Identifikationsroutine.

Außerdem muß bei der Festlegung der Prioritäten für die Peripheriegeräte mit viel Geschick vorgegangen werden. Interrupts sollten nach Möglichkeit nur dort angewandt werden, wo es sich um zeitkritische Anwendungen handelt, die eine schnelle Reaktion des Mikroprozessors erfordern.

5.2.3 Datenaustausch durch direct memory access transfer (DMA)

Diese Art des Datenaustausches wird in Minicomputern sehr häufig angewandt und findet allmählich auch bei Mikroprozessoren Anwendung. Bevor auf einzelne verschiedene Methoden des DMA-Transfers eingegangen wird, sollen die Grundlagen dieses Transfers besprochen werden. In Bild 5.11 ist ein Mikroprozessorsystem mit einem DMA-Interface vorgestellt. An diesem Interface-Baustein fällt auf, daß alle zur Adressierung, zum Datenaustausch und zur Kontrolle eines Speichers ($R/\overline{W}$) notwendigen Signale vom Interface zum Speicher vorhanden sind. Daraus kann eine Eigenschaft des DMA-Transfers abgeleitet werden: Der DMA-Transfer hat es mit dem Datenaustausch zwischen Speicher und Peripheriegerät zu tun. Der Mikroprozessor ist an diesem Transfer *nicht* beteiligt, d.h. der Transfer läuft unter alleiniger Kontrolle des DMA-Interface-Bausteins ab. (Der interne Status des Mikroprozessors muß nicht abgespeichert werden!) Die Auslösung eines DMA-Transfers erfolgt nach Empfang des DMA-Grant-Signals, das vom Mikroprozessor nach Erhalt des DMA-Request-Signals an das Interface abgegeben wird. Sodann wird über das Adressenregister und das Datenregister der Informationsaustausch zwischen Speicher und Peripheriegerät vorgenommen, nachdem zuvor noch die Richtung des Transfers festgelegt wurde, d.h. auslesen (read) oder einschreiben (write) in den Speicher. Um aufeinanderfolgende Speicherplätze zu adressieren, muß das Adressenregister automatisch erhöht oder erniedrigt werden können. Auch muß das Ende eines Transfers über eine Logik innerhalb des Interface-Bausteins festgestellt werden können (word count).

Abschließend sollen zum DMA-Transfer noch einige Bemerkungen über das Verhalten des Mikroprozessors während des Transfers gemacht sowie Anwendungsbeispiele für diesen Datentransfer angegeben werden.

Die erste Methode besteht darin, daß der Mikroprozessor während des DMA-Transfers über ein externes Signal angehalten wird. Dies hat einen Programmstop für die Dauer des Transfers zur Folge. Die mit dem DMA-Transfer erreichbare Datenaus-

tauschrate hängt von der Zykluszeit der verwendeten Speicher ab. Dieses Verfahren hat den Nachteil, daß eine Verzögerung in der Abarbeitung des Programms auftritt.

Mit einer zweiten Methode, dem sogenannten cycle stealing kann obiger Nachteil vermieden werden. Diese Methode arbeitet nach dem Prinzip, das in Bild 5.11 dargestellt ist.

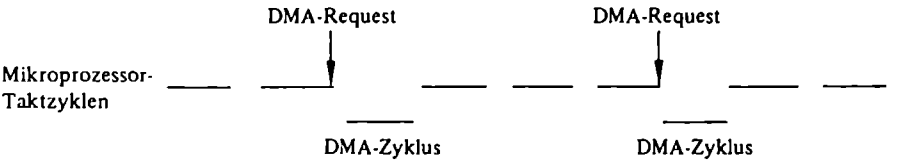
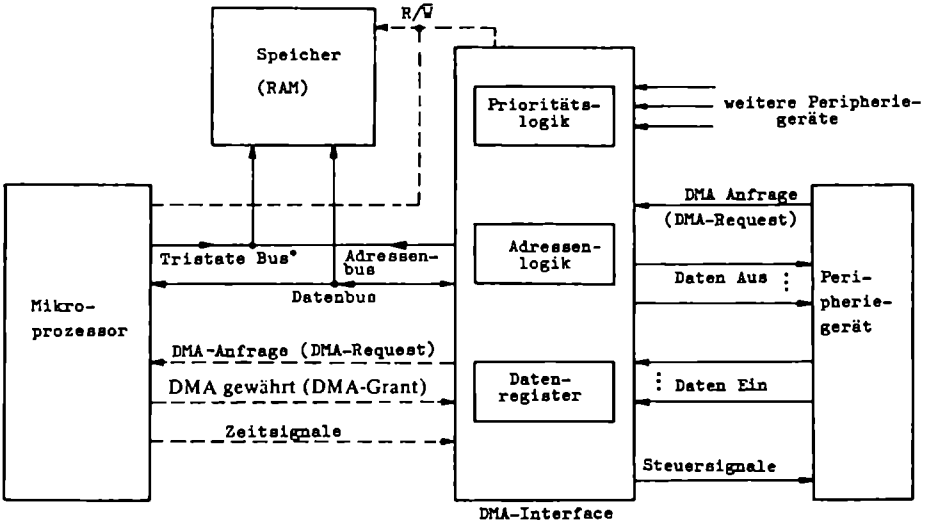


Bild 5.11: Prinzip „Cycle Stealing"

Bild 5.11 zeigt, daß für einen DMA-Transfer Mikroprozessortaktzyklen verwendet („gestohlen") werden, d.h. während dieser Zeit wird das Programm nicht abgearbeitet. Zusammenhängend betrachtet sieht es so aus, als ob der Mikroprozessor mit einem verlangsamten Takt sein Programm abarbeitet.

Ein Datenaustausch durch DMA wird immer dort eingesetzt, wo schnell eine große Datenmenge zwischen Speicher und Peripheriegerät (Platte, Trommel) ausgetauscht werden soll. Das benötigte DMA-Interface ist, verglichen mit den Bausteinen, die für die anderen Datenaustauschverfahren benötigt werden, wesentlich komplizierter (Bild 5.12).



*Wenn kein Tristate-Bus vorliegt, muß durch zusätzliche Logik für den Fall eines DMA-Transfers die Durchschaltung des Adressen- und des Datenbus vom DMA-Interface ermöglicht werden.

Bild 5.12: Datenaustausch durch direct memory access transfer (DMA)

Kapitel 6
Softwarehilfsmittel zur Programmerstellung

6.1 Vereinbarungen und Definitionen

In diesem Kapitel werden die Hilfsmittel vorgestellt, deren sich der Programmierer, der Mikroprozessoranwender, bedient, um zu einer funktionsfähigen Software zu gelangen.

Bevor die einzelnen Hilfsprogramme besprochen werden, müssen für die Programmerstellung gewisse Vereinbarungen und Definitionen gemacht werden.

6.1.1 Vereinbarungen

Erstens:

Aus den vielen Möglichkeiten, Programme zu erstellen, sei eine herausgegriffen: Das Programm soll unter Verwendung eines Entwicklungssystems erstellt werden. Darunter wird ein Mikrocomputer verstanden, bestehend aus Mikroprozessor, Speicher sowie Ein-/Ausgaberegister (Bild 6.1).

Als Eingabe dient eine Teletype, als Ausgabe ein Lochstreifenstanzer.

Zweitens:

Das Programm wird in Assemblersprache, einer mikroprozessorspezifischen symbolischen Programmiersprache, geschrieben.

Drittens:

Die Hilfsprogramme laufen auf dem Entwicklungssystem; sie sind daher ebenfalls in Assemblersprache geschrieben.

Diese Voraussetzungen ermöglichen es, die wesentlichen Phasen, die bei der Programmerstellung durchlaufen werden müssen, deutlich herauszuarbeiten.

Im folgenden werden Definitionen einzelner Begriffe, die im Zusammenhang mit der Programmerstellung von Bedeutung sind, gegeben.

6.1.2 Definitionen

Editor

Der Editor ist ein textverarbeitendes Hilfsprogramm zur Softwareerstellung. Mit ihm kann der Anwender sein zu schreibendes Programm in den Schreib-Lese-Speicher eines Entwicklungssystems eingeben. Spezielle Editorbefehle gestatten außerdem die

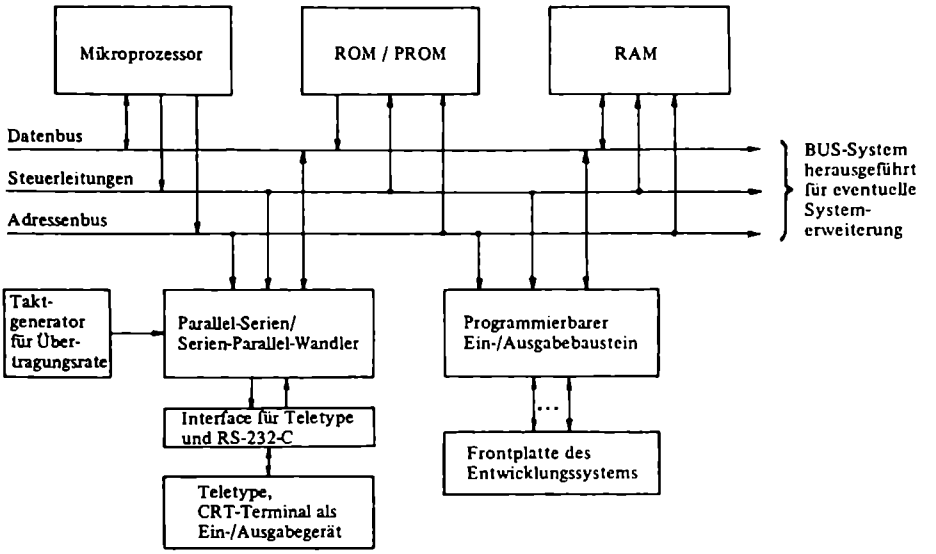


Bild 6.1: Blockschaltbild eines Entwicklungssystems

Ausgabe geschriebener Programme, das Einfügen neuer und die Korrektur fehlerhafter Programmteile. Der Editor (herausgeben engl.: to edit) erleichtert also Programmerstellung und Programmkorrektur wesentlich (Abschnitt 6.2).

Assemblersprache

Unter Assemblersprache, auch Assembliersprache genannt, wird eine mikroprozessorspezifische symbolische Programmiersprache verstanden.

„Symbolisch" besagt, daß zum einen symbolische Befehle, zum anderen symbolische Adressen verwendet werden können. Unter symbolischen Befehlen sind die Mnemonics zu verstehen, d.h. die im Befehlsvorrat für die einzelnen Befehle definierten Abkürzungen. Die Möglichkeit, symbolische Adressen zu verwenden, gestattet dem Programmierer, Adressen innerhalb des Programms sinnvoll zu benennen. Der Anfang eines Programms wird z.B. den Namen START oder INIT erhalten.

Assembler

Unter dem Assembler (zusammensetzen engl.: to assemble) versteht man ein Hilfsprogramm, das ein in Assemblersprache geschriebenes Programm in den Maschinencode des Mikroprozessors übersetzt und dabei jeder Speicheradresse ein Bitmuster als Speicherwort zuordnet.

Die Übersetzung geschieht derart, daß jedem Befehl sein entsprechendes Bitmuster (Maschinencode) zugeordnet wird (Abschnitt 6.3).

Debugging Routine (entwanzen engl.: to debug)
Die Debugging Routine leistet das Austesten des assemblierten Anwenderprogramms. Fehlerhafte Programme können durch Ändern von Speicherstellen korrigiert werden (Abschnitt 6.4).

Im folgenden wird der Weg vom Flußdiagramm zum getesteten, funktionsfähigen Programm in Maschinencode aufgezeigt. Dabei werden im einzelnen der Editor, der Assembler und die Debugging Routine besprochen. Besonders wird auf den Einsatzbereich dieser Hilfsprogramme während der Programmerstellung eingegangen, die in Bild 6.2 dargestellt sind.

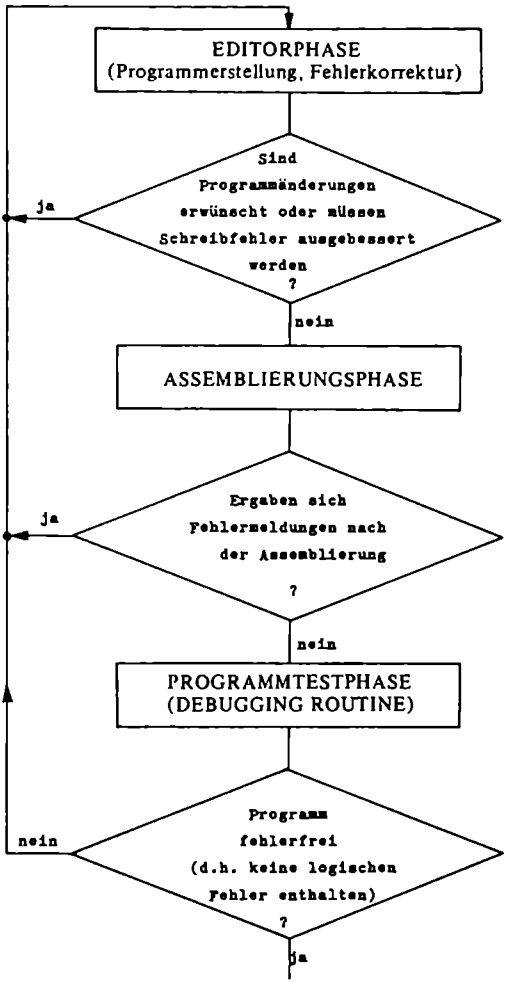


Bild 6.2: Einsatzbereich der Softwarehilfsmittel (schematisch)

6.2 Der Editor

Bei der Programmerstellung geht der Programmierer von einem Flußdiagramm aus, welches die zu lösende Aufgabe in Form kleiner, leicht zu überblickender und nacheinander auszuführender Funktionsschritte darstellt. Ein Beispiel eines solchen Flußdiagramms ist für das mit Hilfe des Editors zu schreibende Programm gezeigt (Bild 6.3).

Damit sich der Entwickler eines Editors bedienen kann, benötigt er ein Computersystem, das über eine Eingabeeinheit (Teletype, Bildschirmterminal) und eine Ausgabeeinheit (Lochstreifenstanzer, Band, Platte) verfügt. Entsprechend den getroffenen Vereinbarungen wird der mit einem Entwicklungssystem (development system) vom Mikroprozessorhersteller angebotene Editor verwendet. Dieser Editor läuft auf dem Entwicklungssystem, das aus einem kompletten Mikrocomputer (CPU, RAM, ROM, I/O-Bausteine) besteht, an das unter anderem eine Teletype, ein Lochstreifenleser und -stanzer sowie ein PROM-Programmiergerät angeschlossen werden können (siehe Bild 6.1).

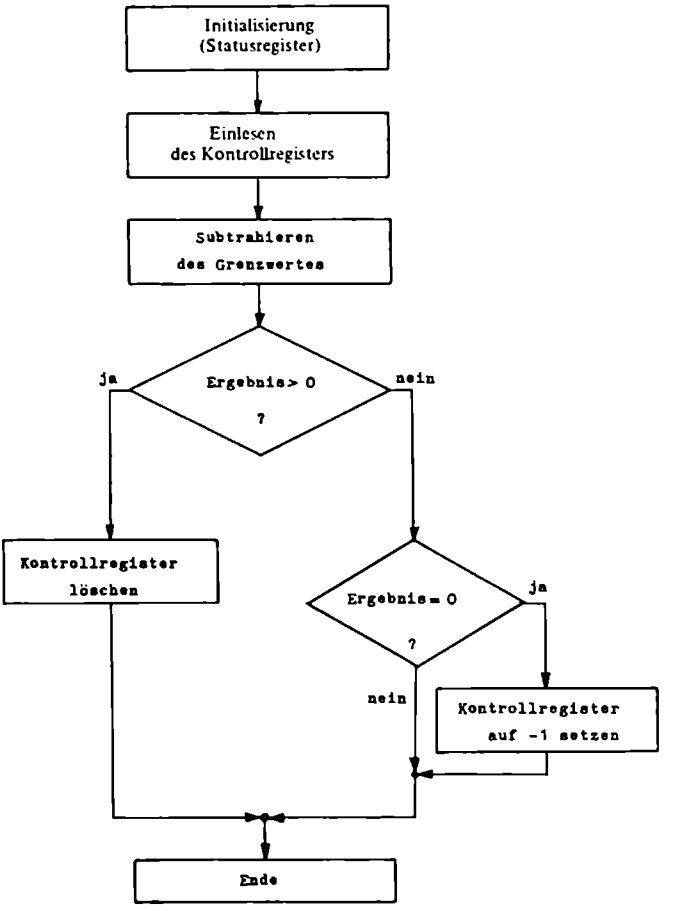


Bild 6.3: Flußdiagramm

Die Arbeitsweise eines Editorprogramms und damit seine Leistungsfähigkeit sollen anhand einiger Beispiele dargestellt werden. Besonders muß darauf hingewiesen werden, daß die von Mikroprozessorherstellern erhältlichen Editorprogramme in ihrer Leistungsfähigkeit stark unterschiedlich sind. Aus diesem Grunde erfolgt die Betrachtung des Editors in verallgemeinerter Form, d.h. nicht spezifisch für einen bestimmten Mikroprozessor.

Zu Beginn dieses Abschnitts wurde erwähnt, daß der Editor ein Hilfsmittel zur Programmerstellung ist. Die zu erstellenden Programme können in jeder Programmiersprache (Fortran IV, Algol, PL/M, Assembler) geschrieben werden, wenn z.B. Zeichen des ASCII-Formats (American Standard Code for Information Interchange) verwendet werden.

Der Editor wird in den Speicher des Entwicklungssystems über den Lochstreifenleser geladen und gestartet.

Bevor mit dem Editor gearbeitet werden kann, muß der Benutzer festlegen, woher die Eingabe erfolgen und wohin das editierte Programm ausgegeben werden soll, d.h. ob es auf Lochstreifen ausgestanzt oder auf Band abgespeichert wird. Im vorliegenden Fall wird der Stanzer als Ausgabeeinheit angegeben.

Handelt es sich wie im vorliegenden Fall um die Erstellung eines vollkommen neuen Programms, so ist als Eingabe die Teletype anzugeben. Soll ein bereits vorhandenes Programm mit dem Editor umgeändert, ergänzt oder korrigiert werden, so muß zusätzlich noch als Eingabe die Einheit angegeben werden, auf der das zu behandelnde Programm abgespeichert ist, um es von dorther einzulesen.

Der Editor wartet nun auf die Befehle des Benutzers, die dieser z.B. über die Teletype eingibt. Dabei ist folgendes zu beachten: Der Editor kennt zwei verschiedene Zustände:

- Befehlszustand,
- Textzustand.

Der Befehlszustand

Befindet sich der Editor in diesem Zustand, so interpretiert er vereinbarte Zeichen, die über die Teletype eingegeben werden, als *Befehle*. Einige charakteristische Befehle werden im weiteren besprochen.

Der Textzustand

Befindet sich der Editor in diesem Zustand, so werden alle über die Teletype eingegebenen Zeichen als *Text* interpretiert und im Arbeitsbereich (text buffer) des Editors abgelegt. Dieser Arbeitsbereich (RAM) ermöglicht, eine durch seine Speichergröße begrenzte Anzahl von Zeichen oder Zeilen aufzunehmen. Dabei wird jeder Zeile über einen vom Editorprogramm gesteuerten Zähler eine Nummer zugeordnet, auf die mit den Editor-Befehlen Bezug genommen werden kann, sofern sich dieser im Befehlszustand befindet.

6.2.1 Arbeitsweise des Editorprogramms

Der Editor wurde geladen und gestartet. Weiterhin wurden Angaben gemacht, die die Eingabe und Ausgabe des Programms betrafen. Damit ist der Editor in der Lage, gemäß den Befehlen, die über die Teletype eingegeben werden, zu handeln.

Die vorgegebene Aufgabe bestand darin, das durch ein Flußdiagramm festgelegte Programm zu schreiben, zu „editieren".

Der Editor erhält den Befehl, daß Text über die Teletype eingegeben werden möchte, indem ein vereinbarter Buchstabe auf der Teletype gedrückt wird. Der Editor liest danach jeden auf der Teletype gedrückten Buchstaben in den „text buffer" ein und erhöht mit dem Beginn jeder neuen Zeile den Zeilenzähler („buffer pointer") um Eins. Auf diese Weise wird jeder Programmzeile eine Nummer zugeordnet, wodurch bei späteren Korrekturen die Möglichkeit der schnellen Bezugnahme eines Editor-Befehls auf eine fehlerhafte Zeile gewährleistet ist.

Die Vorteile des Editors kommen in dem Augenblick zur Geltung, wenn während des Editierens eines Programms oder nach dem Testen desselben Fehler zu korrigieren sind. In diesem Fall wird die Teletype mit Hilfe des Editors zu einem sehr leistungsfähigen Eingabegerät. Ohne Editor müßten alle notwendigen Programmänderungen am Lochstreifen, auf dem sich das Programm befindet, in mühseliger und zeitraubender Kopier- und Korrekturarbeit manuell ausgeführt werden.

Bild 6.4 zeigt ein in Assemblersprache geschriebenes Programm. Am Beispiel dieses editierten Programms seien einige charakteristische Editor-Befehle vorgestellt.

```
/ Programm zum Testen eines Interface-Kontrollregisters, das mit KONTR bezeichnet wird.
/ Es wird getestet mit einem Grenzwert, der am Speicherplatz LIMIT abgelegt ist.
/ Es wird der Befehlsvorrat des hypothetischen Mikroprozessors verwendet.

          ORG 200          / Startadresse

          KONTR EQU 4000

INIT,     CLRA             / 0 → ACCA
          TAP              / 0 → CCR
          LDAA E KONTR     / Einlesen des Kontrollregisters
          SUBA E LIMIT
          BGT  GRZERO      / Ergebnis > 0, Verzweigung zu GRZERO
          BEQ  EQZERO      / Ergebnis = 0, Verzweigung zu EQZERO
          BRA  STOP

GRZERO,   CLR E KONTR      / Löschen des Kontrollregisters
          BRA  STOP

EQZERO,   LDAA # 377       / Setzen des Akkumulators auf −1
          STAA E KONTR     / Setzen des Kontrollregisters auf −1

STOP,     BRA  STOP        / Ende des Programms

LIMIT,    200              / Grenzwert = 200 8

          END              / Ende des Assembliervorgangs
```

Bild 6.4: Quellenprogramm in Assemblersprache

- Auslisten eines Teils oder des gesamten Programms auf der Teletype. Der gewünschte, auszulistende Bereich kann in Form von Grenzen angegeben werden (L; nL; m, nL).
- Auslöschen eines Teils oder des gesamten Programms (D; nD; m, nD).
- Einfügen neuer Programmteile vor einer angegebenen Zeile (nI). Nach Erhalt dieses Befehls befindet sich der Editor im Textzustand. Ist das Hinzufügen neuer Programmteile beendet, so muß der Editor wieder in den Befehlszustand zurückgebracht werden.
- Ersetzen einer Zeile durch eine oder mehrere neue Zeilen (nC; m, nC).
- Suchen nach einem bestimmten Buchstaben innerhalb einer Zeile. Dies ist dann nützlich, wenn ein oder mehrere Fehler innerhalb einer Zeile vorkommen. Dieser Befehl ermöglicht es, gezielte Änderungen an Teilen dieser Zeile auszuführen (nS).

  So zum Beispiel:

  Korrektur eines Schreibfehlers,
  Einfügen von Kommentar,
  Auslöschen eines falschen Befehls,
  Verbesserung von Syntaxfehlern.

- Suchen nach einem bestimmten Wort innerhalb des Arbeitsbereiches des Editors. Durch den Befehl definiert der Benutzer ein Suchwort, auf das hin der Editor das Programm durchsucht. Nach erfolgreicher Suche können *in* der betreffenden Zeile oder in einem *angegebenen Bereich* des Suchwortes Änderungen durchgeführt werden (Auslöschen, Ersetzen, Auslisten).

  Diese Art des Suchens mit Hilfe eines Suchwortes kann bei manchen Editorprogrammen auch angewandt werden, wenn sich das gewünschte Suchwort nicht innerhalb des Arbeitsbereiches befindet, d.h. das editierte Programm ist größer als der zur Verfügung stehende Arbeitsbereich des Editors. Es muß vom Editor der bereits durchsuchte Arbeitsbereich zur festgelegten Ausgabeeinheit transferiert und ein neuer Programmteil in den vorher gelöschten Arbeitsbereich eingelesen werden. Danach beginnt der Suchvorgang von neuem. Dies wird so lange fortgesetzt, bis entweder das Suchwort gefunden wurde oder kein weiterer Programmteil mehr zur Verfügung steht.

- Sehr nützlich und des öfteren auch schon bei Editorprogrammen für Mikroprozessoren anzutreffen ist ein sogenannter kombinierter Such- und Ersetzbefehl.

  Dieser Befehl funktioniert wie der vorher behandelte, nur mit der Erweiterung, daß zusätzlich zum Suchwort ein weiteres sogenanntes Ersatzwort definiert wird. Dieser Befehl wirkt nun so, daß für den Fall eines erfolgreichen Suchens an der Stelle des gefundenen Suchwortes das Ersatzwort eingesetzt wird. Es erfolgt somit ein Austausch zweier Worte bzw. Satzteile.

- Oft ist es von Nutzen, wenn ein bestimmter Befehl, wie z.B. ein Suchbefehl, mehrmals hintereinander ausgeführt wird, um häufig wiederkehrende Fehler schnell zu korrigieren.

  Der mehrmalige Ablauf eines Befehls ist auch für den bereits erwähnten Such- und Ersetzbefehl von großer Bedeutung. Dem Benutzer wird dadurch eine schnelle Änderung derselben fehlerhaften oder verbesserungswürdigen Stellen ermöglicht.

- Wichtige Editorbefehle sind weiterhin solche, die eine Kombination verschiedener Befehle ermöglichen, z.B. Suchen – Ersetzen; Suchen – Auslöschen; Ausgeben des bearbeiteten Arbeitsbereiches – Einlesen eines neuen Arbeitsbereiches.
- Will der Benutzer seine Programmteile völlig umstellen, so bedient er sich vorteilhafterweise eines Befehls, der eine Verschiebung einzelner Programmteile gestattet. Angegeben werden müssen dabei die beiden Grenzen des zu verschiebenden Bereiches und die Stelle, vor die der angegebene Bereich zu bewegen ist.
- Nach Abschluß des Editionsvorgangs muß die Möglichkeit bestehen, das ausgebesserte, umgeänderte bzw. neugeschriebene Programm auf eine Ausgabeeinheit auszugeben (Lochstreifenstanzer), um es für eine spätere Assemblierung abzuspeichern. Für den Lochstreifenstanzer müssen, wenn bestimmte Anforderungen an das Lochstreifenformat bezüglich Vorlauf (leader) und Nachlauf (trailer) gestellt werden, diesbezüglich Befehle vorhanden sein.

Zusammenfassend läßt sich über den Editor sagen: Aus den angegebenen Beispielen eines Editors läßt sich ersehen, daß durch dieses Hilfsprogramm die Programmerstellung, vom Flußdiagramm (Bild 6.5) ausgehend, bis hin zur Korrektur der beim Austesten festgestellten Fehler, erheblich erleichtert wird. Zudem stellt der Editor mit den vorhandenen Befehlen für Eingabe, Ausgabe und die Durchführung der anfallenden Korrekturen ein Unterstützungsprogramm dar, welches dem Benutzer wertvolle Dienste bei der Erstellung seines für die spätere Assemblierung benötigten Quellenprogramms (source program) leistet.

Phase der Programmerstellung	Zur Verfügung stehendes Gerät	Verwendete Software	Ergebnis
Editorphase			
Schreiben des Quellenprogramms in Assemblersprache, ausgehend vom Flußdiagramm	**Eingabe-Einheit** Teletype Terminal	Editorprogramm eines Entwicklungssystems	Quellenprogramm (source program) auf Lochstreifen, Band oder Platte abgespeichert
Korrektur bereits geschriebener Programme, die Fehler nach der Assemblierung oder beim Austesten („Debugging") zeigten			

Bild 6.5: Einsatzbereich des Editors

6.3 Der Assembler

Im letzten Abschnitt wurde gezeigt, wie mit Hilfe des Editors ein Programm geschrieben (editiert) werden kann[1]). Mit dem Assembler wird ein weiteres Hilfsmittel vorgestellt, das den Programmierer in die Lage versetzt, sein mit dem Editor erstelltes Programm in ein für den Mikroprozessor verständliches Format, das Binärformat, zu übersetzen. Anhand von Programmbeispielen, die mit dem aus Kapitel 4 bekannten Befehlsvorrat des hypothetischen Mikroprozessors geschrieben wurden, wird die Wirkungsweise eines Assemblers erklärt und auf seine Vorzüge gegenüber manuellen Assemblierungsmethoden eingegangen. Es sei betont, daß der vorgestellte Assembler nicht von einem Hersteller angeboten wird. Dieser Abschnitt behandelt somit in allgemeiner Form ein Hilfsmittel zur Softwareerstellung, den Assembler.

Der Programmierer hat mehrere Möglichkeiten, sein Programm zu schreiben. Er kann es in dualer, oktaler, hexadezimaler Form und in Assemblersprache[2]) tun. Davon ist nur die Programmdarstellung in Assemblersprache heute üblich und gebräuchlich. Eine Programmdarstellung in dualer, oktaler oder gar hexadezimaler Form hat, wie aus den Bildern 6.6a - 6.6c hervorgeht, entscheidende Nachteile, denn diese Darstellungen besitzen folgende Unzulänglichkeiten:

Sie sind unübersichtlich; zeitraubend zu erstellen, da sehr fehleranfällig; für einen anderen Anwender unverständlich.

Diese Unzulänglichkeiten, die vor allem bei der dualen Darstellung auftreten, werden auch durch Verwendung der oktalen bzw. hexadezimalen Darstellung nicht umgangen.

Aus den genannten Gründen, und vor allem um eine bessere Verständlichkeit für andere Anwender zu ermöglichen und um das Testen des Programms zu erleichtern, wurde die Darstellung eines Programms in Assemblersprache geschaffen.

Als Assemblersprache bezeichnet man die Summe der Befehle (Befehlsvorrat), die in Form von Mnemonics für einen Mikroprozessor zur Verfügung stehen. Die *Assemblersprache* ist somit für jeden Mikroprozessor verschieden, d.h. sie ist *mikroprozessorspezifisch*. Ein Programm in Assemblersprache stellt eine Folge von Mnemonics dar. Diese sind, wie aus Kapitel 4 bereits bekannt ist, eine Abkürzung für einen bestimmten Befehl. Ein Befehl teilt dem Mikroprozessor über ein bestimmtes binäres Format mit, welche Operation er auszuführen hat. Es gilt somit der folgende Zusammenhang:

Mnemonic → Befehl → binäres Format

Dieser Zusammenhang macht deutlich, daß zwischen Mnemonic und dem entsprechenden binären Format eines Befehls eine feste Relation besteht. Ein Mikroprozessor benötigt, um einen Befehl ausführen zu können, den Binärwert eben dieses Befehls. Aus diesem Grund muß eine Übersetzung des Programms und damit der Mnemonics (symbolischer Befehl, symbolischer Assembler) in das entsprechende binäre

[1]) In Verbindung mit dem Assembler handelt es sich bei dem Geschriebenen zweckmäßigerweise um ein Programm in Assemblersprache.

[2]) Es seien hier, gemäß den Vereinbarungen, die Hochsprachen (z.B. Fortran IV) nicht betrachtet.

Dual	10 000 000	10 110 010
	10 000 001	10 000 101
	10 000 010	10 010 001
	10 000 011	00 001 000
	10 000 100	00 000 000
	10 000 101	00 100 001
	10 000 110	00 000 000

Bild 6.6a: Duale Codierung

Oktal	200	262
	201	205
	202	221
	203	010
	204	000
	205	041
	206	000

Bild 6.6b: Oktale Codierung

Hexadezimal	80	B2
	81	85
	82	91
	83	8
	84	0
	85	21
	86	0

Bild 6.6c: Hexadezimale Codierung

Format durchgeführt werden. Diese Übersetzung ist *eine* Aufgabe, die der Assembler zu leisten hat. Der Assembler führt eine *eins-zu-eins-Übersetzung* durch, d.h. einem Mnemonic entspricht ein binäres Äquivalent.

Das in Assemblersprache vorliegende Programm wird als „source program", das vom Assembler übersetzte und vom Mikroprozessor ausführbare wird als „object program" bezeichnet (Bild 6.7).

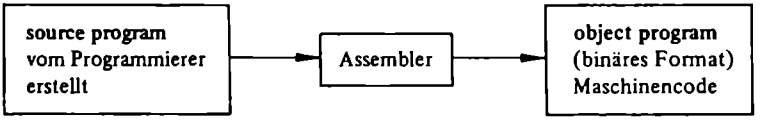


Bild 6.7: Übersetzung des source program ins object program (Maschinencode)

Wie geht der Assembler bei der Übersetzung vor und was leistet er noch darüber hinaus?

Diese Frage soll mit Hilfe eines in Assemblersprache geschriebenen Programms geklärt werden. Dazu sei nochmals auf Bild 6.4, das mit dem Editor geschriebene Programm, Bezug genommen.

```
/ Programm zum Testen eines Interface-Kontrollregisters, das mit KONTR bezeichnet wird.
/ Es wird getestet mit einem Grenzwert, der am Speicherplatz LIMIT abgelegt ist.
/ Es wird der Befehlsvorrat des hypothetischen Mikroprozessors verwendet.

0200            ORG                             / Startadresse
4000            KONTR  EQU 4000
```

Speicher-adresse (oktal)	Speicher-inhalt (oktal) (Maschinen-code)	Etikett-Feld (label) (symbolische Adresse)	Mnemonic (Code-Feld)	Operand-Feld (Operand)	Kommentar-Feld
00200	0262	INIT,	CLRA		/ 0 → ACCA
00201	0205		TAP		/ 0 → Statusregister (CCR)
00202	0221		LDAA	E KONTR	/ Einlesen des Kontrollregisters
00203	0010				
00204	0000				
00205	0041		SUBA	E LIMIT	
00206	0000				
00207	0232				
00210	0376		BGT	GRZERO	/ Erg. >0, Verzweigung zu GRZERO
00211	0004				
00212	0367		BEQ	EQZERO	/ Erg.=0, Verzweigung zu EQZERO
00213	0007				
00214	0361		BRA	STOP	
00215	0012				
00216	0276	GRZERO,	CLR	E KONTR	/ Löschen des Kontrollregisters
00217	0010				
00220	0000				
00221	0361		BRA	STOP	
00222	0005				
00223	0220	EQZERO,	LDAA #377		/ Setzen des Akkumulators auf −1
00224	0377				
00225	0241		STAA	E KONTR	/ Setzen des Kontrollregisters
00226	0010				/ auf −1
00227	0000				
00230	0361	STOP,	BRA	STOP	/ Ende des Programms
00231	0376				
00232	0200	LIMIT,	200		/ Grenzwert = 200_8
			END		

Zuordnung von symbolischen Adressen zu Speicheradressen (Symboltafel, symbol table).

EQZERO	223
GRZERO	216
INIT	200
KONTR	4000
LIMIT	232
STOP	230

Bild 6.8:
Programmliste und Symboltafel nach Assemblierung

Die Assemblersprache gehorcht wie jede andere Sprache gewissen Regeln des Satzbaus (Syntax). Der Assembler prüft die Einhaltung dieser Regeln nach. Jeder Zeile sind vier Felder zugeordnet:

Etikett-	(label)	Feld
Code-	(code)	Feld
Operand-	(operand)	Feld
Kommentar-	(comment)	Feld

Zur Erkennung für den Assembler wird jedes Feld von dem benachbarten durch ein spezielles vereinbartes Zeichen (, /) bzw. einen definierten freien Zwischenraum getrennt (siehe Bild 6.8).

Der Assembler liest das z.B. auf Lochstreifen befindliche Quellenprogramm (source program) ein. Die Übersetzung des Quellenprogramms und die Zuordnung des Maschinencodes (object code) zu bestimmten Adressen erfolgt in mehreren Durchgängen (pass), meistens 2 oder 3. Nach der Anzahl der Durchgänge wird der betreffende Assembler als Zwei-Pass- bzw. Drei-Pass-Assembler bezeichnet.

Im ersten Durchgang stellt der Assembler eine Tafel von symbolischen Adressen auf (symbol table) und prüft zugleich das „source program" auf etwaige Syntaxfehler nach.

Erstellung der Symboltafel (symbol table)

Durch die Assembleranweisung (Assembler directive) ORG 200 weiß der Assembler, daß er der ersten Speicheradresse den Wert 200 zuzuordnen hat. Er setzt einen internen Zähler auf 200 und erhöht diesen abhängig vom eingelesenen Mnemonic um Eins, Zwei oder Drei, je nachdem, ob es sich um einen Ein-, Zwei- oder Drei-Byte-Befehl handelt. Dies weiß der Assembler aufgrund seiner internen Liste, in der alle Mnemonics des Befehlsvorrats, einschließlich der damit möglichen Adressierungsarten, wie in einem Lexikon abgelegt sind. Kommt er bei diesem Vorgang auf ein besetztes Label-Feld, d.h. eine symbolische Adresse, so weist er dieser den laufenden Wert des Zählers zu. Auf diese Weise erhält der Assembler eine Liste von symbolischen Adressen (Symboltafel oder symbol table), denen ein bestimmter fester Wert, relativ zu ORG 200, zugeordnet ist. Für unser Beispiel lautet diese Liste in alphabetischer Reihenfolge:

EQZERO	223
GRZERO	216
INIT	200
KONTR	4000
LIMIT	232
STOP	230

Bild 6.9: Symboltafel

Damit ist der erste Durchgang abgeschlossen, und wenn keine Fehlermeldung ausgegeben wurde, kann der zweite Durchgang begonnen werden. Obige Liste wird zusätzlich zur Liste der Mnemonics als Tafel vom Assembler weiter benützt.

Bevor zum zweiten Durchgang übergegangen wird, sollen noch einige Anmerkungen zur Verwendung von *symbolischen Adressen* gemacht werden. Die Möglichkeit der symbolischen Adressierung erleichtert dem Programmierer die Arbeit beträchtlich, denn er ist in der Lage, beliebigen Sprungadressen oder Speicherzellen, die beson-

dere Konstanten erhalten, signifikante Namen zu geben. In unserem Beispiel: GRZERO, EQZERO und LIMIT. Ohne diese Möglichkeit müßte z.B. eine Sprungadresse vom Programmierer selbst berechnet werden, was bei größeren Programmen umständlich und mühsam sein kann. Die symbolische Adressierung erleichtert das Testen und Lesen eines Programms erheblich. Da eine Sprungadresse von der Anfangsadresse eines Programms abhängig ist (ORG 200!), müßte bei der Verwendung expliziter Adressen (z.B. JMP 100 (**Jump** 100)) jedesmal die Sprungadresse vom Programmierer neu berechnet werden, wenn das Programm an einer anderen Stelle assembliert wird. Dieser Nachteil wird durch die Verwendung symbolischer Adressen umgangen, denn diesen ordnet der Assembler einen Wert relativ zur Anfangsadresse des Programms (ORG) zu.

Im zweiten Durchgang erfolgt die Übersetzung (Assemblierung) des Quellenprogramms in Maschinencode. Der Assembler geht dabei folgendermaßen vor:

Er liest z.B. die Zeile SUBA E LIMIT ein.
Er vergleicht das Mnemonic SUBA mit seiner internen Liste und liest weiterhin das Symbol E ein, welches als vereinbartes Zeichen für die extended Adressierung steht.

Da dieses Mnemonic SUBA und die angegebene Adressierungsart E gültig sind, wird vom Assembler für SUBA E das binäre Äquivalent 00100001_2 (41_8) eingesetzt und in der Adresse 205 abgelegt. Damit ist jedoch nur ein Teil des Befehls abgearbeitet. Es fehlt noch die Übersetzung der symbolischen Adresse LIMIT. Der Assembler durchsucht seine Symboltafel und stellt fest, daß der symbolischen Adresse LIMIT der Wert 10011010_2 (232_8) zugeordnet ist und daß diese Adresse mit extended Adressierung zu erreichen ist. Da es sich um einen Drei-Byte-Befehl handelt, wird dieser Wert (232_8) den Speicherplätzen 206 und 207 als Inhalt zugeordnet (Bild 6.8). Die Zeile SUBA E LIMIT ist somit vom Quellenprogramm in Maschinencode übersetzt worden (Bild 6.8). Der Assembler setzt diesen Vorgang so lange fort, bis er auf die Assembleranweisung END stößt, die ihm mitteilt, daß das Quellenprogramm zu Ende ist. Der Assembler hat somit den zweiten Durchgang beendet.

Parallel zum Assemblierungsvorgang im zweiten Durchgang wird vom Assembler über ein Ausgabegerät ein Lochstreifen mit dem Maschinencode ausgegeben und zudem noch, sofern beide Ausgabegeräte getrennt sind, die Symboltafel und eine Liste, die nebeneinander das assemblierte und das ursprüngliche Quellenprogramm enthält[1]). Diese Liste ist für eine spätere Dokumentation sehr wichtig und leistet zudem noch beim Testen des Programms wertvolle Dienste, da aus ihr eine direkte Zuordnung zwischen Speicherzelle, deren Inhalt und des Mnemonics sowie eines möglicherweise eingefügten Kommentars zu ersehen ist (Bild 6.8).

Neben der Übersetzung von Quellenprogrammen in Maschinencode besitzt ein Assembler häufig weitere nützliche Fähigkeiten:

1. Prüfung auf Syntaxfehler und entsprechender Fehlermeldungen. (*Keine* Prüfung auf logische Fehler).
2. Verwendung sogenannter Pseudobefehle.
3. Möglichkeit der bedingten Assemblierung (conditional assembly).
4. Möglichkeit zur Definition von Makros.

[1]) Wenn dies nicht der Fall ist, so muß zur Ausgabe der Liste bzw. des Lochstreifens mit dem Maschinencode ein dritter Durchgang erfolgen.

Zu Punkt 1

Wie bereits erwähnt, stellt der Assembler im ersten Durchgang eine Symboltafel auf und prüft zusätzlich das Quellenprogramm, z.B. auf ordnungsgemäße Syntax, durch Vergleich mit den definierten Mnemonics, Adressierungsarten und Zahlensystemen nach. Beides ist notwendig, um eine spätere fehlerfreie Übersetzung des Quellenprogramms in Maschinencode zu ermöglichen.

Wird während des ersten Durchgangs ein Fehler gefunden, so gibt der Assembler dies mittels einer entsprechenden Fehlermeldung bekannt, welche die fehlerhafte Zeile und die Art des Fehlers enthält.

Einige charakteristische Fehlermeldungen werden ausgegeben für die folgenden Fälle:

– *Falscher Ausdruck*

  Wenn ein Mnemonic verwendet wird, das der Assembler nicht in seiner Liste vorfindet
  (z.B. BAQ anstatt BEQ).

– *Falsches Befehlsformat*

  Wenn für einen Befehl nicht die festgelegte Anzahl von Operanden angegeben wird (z.B. LDAA # anstatt LDAA # 377).

– *Mehrfache Definition*

  Wenn zwei symbolischen Adressen derselbe Name gegeben wird, d.h. der Assembler nicht imstande ist, einer symbolischen Adresse eindeutig einen Zahlenwert zuzuordnen
  (z.B. es wird zweimal die symbolische Adresse GRZERO verwendet, anstatt zum einen GRZERO und zum anderen EQZERO).

– *Fehlende Definition*

  Wenn innerhalb des Programms eine symbolische Adresse verwendet wird, zu der gesprungen bzw. verzweigt werden soll, deren Zahlenwert dem Assembler aber nicht bekannt ist, weil er die symbolische Adresse nicht in der Symboltafel findet (z.B. BGT GRZERO, aber GRZERO ist nicht als symbolische Adresse definiert).

– *Falsche Ziffer*

  Wenn innerhalb eines definierten Zahlensystems eine Ziffer verwendet wird, die in demselben nicht definiert ist, bzw. eine Zahl dargestellt werden soll, die außerhalb des vereinbarten Bereiches liegt
  (z.B. Oktalsystem 329_8 → 9 keine Oktalzahl oder 8-Bit-Zahlenbereich vereinbart, damit liegt 476_8 außerhalb des Zahlenbereiches).

– *Fehlerhafte Adressierungsart*

  Wenn abweichend von den vereinbarten Kennbuchstaben für die Adressierungsarten andere verwendet werden, falls ein Operand nicht mit der angegebenen Adressierungsart erreicht werden kann oder falls eine verwendete Adressierungsart mit dem betreffenden Befehl nicht verwendet werden kann
  (z.B. LDAA 377 anstatt LDAA # 377).

– *Symboltafel voll*

  Wenn zu viele neue Symbole definiert werden, denen der Assembler einen Wert zuordnen und sie in der Symboltafel ablegen muß, kann es vorkommen, daß der zur Verfügung stehende Bereich nicht ausreicht.

– *Fehlerhafte Syntax*
  Wenn eine Trennung zwischen Etiketten-Feld (label field), Code-Feld sowie Operanden-Feld vergessen wurde.

Für alle angegebenen Fälle gibt der Assembler eine Fehlermeldung an das Ausgabegerät aus.

Zu Punkt 2

Pseudobefehle generieren im Unterschied zu den eigentlichen Befehlen des Mikroprozessors keinen Maschinencode. Sie sind Anweisungen für den Assembler während des Assemblierungsvorgangs und besitzen keinen Einfluß auf das assemblierte Programm.

Einige wichtige Pseudobefehle sind:

– Befehle zur Festlegung des Beginns und des Endes eines Quellenprogramms (z.B. ORG und END).
  Der Befehl ORG bewirkt, daß der interne Zähler des Assemblers auf eine bestimmte Zahl, die Anfangsadresse des Objektprogramms, eingestellt wird (z.B. ORG 200 → Anfangsadresse 200_8).
  Jedes Quellenprogramm muß mit einem Befehl abgeschlossen sein, der dem Assembler mitteilt, daß es zu Ende ist (z.B. END).
– Befehl, der dem Assembler mitteilt, einen zusammenhängenden Block von Speicherzellen auf einen gewünschten Wert, meistens Null, zu setzen. Dabei ist die Länge des Blocks und seine Anfangsadresse zu definieren (z.B. RESULT BLOCK N).
– Befehl, um einem Symbol einen bestimmten Wert zuzuordnen (z.B. VALUE EQU 4
  oder BEGIN EQU A+B).
– Befehle zur Definition von Zahlenbasen (Dezimal, Oktal, Hexadezimal) (z.B. HEX; OCT) und zur Festlegung, ob Einfach- oder Doppelgenauigkeitszahlen (z.B. DOUBLE) gewünscht werden.
– Befehle, die eine bedingte Assemblierung gestatten (z.B. IFZERO, IFNEG, IFPOS, ENDIF) (s. Punkt 3).
– Befehle, die die Bildung von Makros gestatten (z.B. MACRO, MACEND) (s. Punkt 4).

Zu Punkt 3

Die bedingte Assemblierung gestattet die Übersetzung eines Quellenprogrammabschnitts abhängig von einer festgelegten Bedingung, die zur Zeit der Assemblierung besteht. Der betreffende Programmabschnitt ist zwischen zwei Terminatoren eingefügt, die Beginn und Ende dieses Abschnitts bestimmen; weiterhin ist durch sie die Bedingung festgelegt, von der die Assemblierung abhängt. Außerdem kann ein Symbol oder ein algebraischer Ausdruck angegeben sein, auf den die Bedingung zutreffen muß

(z.B. IFZERO VALUE
  LDAA # 252
  TAP /Setzen des Statusregisters abhängig von VALUE
  ENDIF).

Trifft die Bedingung zu (Value = 0), so erfolgt die Assemblierung des angegebenen Programmabschnitts, ansonsten unterbleibt diese.

Zu Punkt 4

Besitzt ein Assembler die Möglichkeit, Makros zu verarbeiten (Makroassembler), so bedeutet dies eine Menge von Vorteilen für den Benutzer. Er kann neue Befehle definieren, die für eine Gruppe von Befehlen in Assemblersprache stehen. Um dies zu tun, gibt er einem Makro einen Namen, nämlich den des gewünschten neuen Befehls, und definiert ihn innerhalb des von den Pseudobefehlen MACRO und MACEND bestimmten Bereiches (macrobody). Benötigt dieser neue Befehl einen oder mehrere Operanden, so müssen diese als Argumente dem Makro bekannt sein (dummy arguments), um sie beim Anruf des Makros in den „Macrobody" an der entsprechenden Stelle einsetzen zu können.

Ein Beispiel soll dies verdeutlichen:
Es soll ein Makro geschrieben werden für die Multiplikation einer Zahl mit 2.

```
MACRO MULT X
LDAA E X        /Einlesen der Zahl (X)
ROLA            /Multiplikation mit 2
MACEND
```

In diesem Beispiel ist MULT der Name des Makros und X sein Argument. An jeder Stelle im Programm kann nun das Macro MULT mit zugehörigem Argument stehen. Anstelle dieses Makros werden nun, vorausgesetzt das Makro wurde im Programm definiert, die im „Macrobody" angegebenen Befehle eingesetzt und assembliert.

Aus diesem kleinen Beispiel läßt sich erkennen, daß Makros ein Weg zu einer übersichtlichen Programmierung sind, weil Funktionen, die nur durch eine Reihe von Befehlen realisierbar sind, mit einem Ausdruck, dem Namen des Makros, dargestellt werden können.

Werden Befehlsgruppen in einem Programm des öfteren wiederholt, so empfiehlt sich die Anwendung von Makros (→ open subroutine, Abschnitt 4.9).

Ein weiterer Grund für den Einsatz von Makros ist die Tatsache, daß sich mit ihrer Hilfe neben Befehlen auch Adressierungsarten eines fremden Mikroprozessors auf einem vorhandenen simulieren lassen.

Phase der Programmerstellung	Zur Verfügung stehendes Gerät	Verwendete Software	Ergebnis
Assemblierungsphase			
Assemblierung eines Quellenprogramms in Assemblersprache	**Entwicklungssystem**	Assembler eines Entwicklungssystems (self oder resident assembler)	Objekt Programm (object program) Programm in Maschinencode auf Lochstreifen bzw. Band *falls gewünscht:* Programmliste und Symboltafel auf Ausgabeeinheit

Bild 6.10: Einsatzbereich des Assemblers

6.4 Die Debugging Routine

Nach Beendigung des Editionsvorgangs steht ein Programm in ASCII-Format auf einem Lochstreifen oder auf Band zur Verfügung. Dieses Programm, auch Quellenprogramm genannt, muß assembliert, d.h. in Maschinencode umgewandelt werden. Wenn dieser Vorgang *ohne* Fehlermeldung von seiten des Assemblers abgeschlossen wurde, steht dem Anwender das Programm in Maschinencode z.B. auf einem Lochstreifen zur Verfügung. Falls gewünscht, besitzt er außerdem eine Programmliste (program listing) (Bild 6.8). Diese Liste enthält die Symboltafel und das assemblierte Programm, d.h. Speicherzellen und deren Inhalt, sowie die dazugehörenden Mnemonics, also das mit Hilfe des Editors in Assemblersprache geschriebene Programm. Für das Austesten des Programms stellt diese Liste ein wertvolles Hilfsmittel dar, da mit ihrer Hilfe das Programm nachvollzogen werden kann.

6.4.1 Austesten eines editierten und assemblierten Programms

Für das Testen des assemblierten Programms stehe ein Entwicklungssystem, wie es von vielen Mikroprozessorherstellern angeboten wird, zur Verfügung. Das zum Testen verwendete Programm des Entwicklungssystems ist Teil seines Monitorprogramms, das zusätzlich zum Testen noch das Laden von Programmen in Maschinencode und deren Ausgabe auf Lochstreifen ermöglicht. Das Monitorprogramm ist für die Überwachung und Kontrolle des Betriebszustandes des Entwicklungssystems zuständig. Der Benutzer hat über ein Eingabegerät (Teletype) die Möglichkeit, dem Monitorprogramm Befehle zu geben.

Das Testen des assemblierten Programms wird auch als „program debugging" bezeichnet und das verwendete Programm dementsprechend als „Debugging Routine". Wie bereits erwähnt, ist diese „Debugging Routine" Teil des Monitorprogramms. Das Testen des Programms erfolgt mit Befehlen, die über das Eingabegerät an das Monitorprogramm gegeben werden (on-line debugging).

Da der Assembler ein Programm nicht auf logische Fehler hin überprüft, ist der Benutzer auf eine Prüfung der logischen Funktionen seines Programms angewiesen. Dies kann er anhand der zur Verfügung stehenden Programmliste tun, indem er das Programm selbst Schritt für Schritt durchprüft. Diese manuelle Prüfung läßt sich bei kleinen Programmen durchführen, ist jedoch für umfangreiche Programme nicht zu empfehlen, da sie zu zeitraubend und fehleranfällig ist.

Das zu testende Programm wird über ein Ladeprogramm in den Speicher des Entwicklungssystems geladen und durch das Monitorprogramm an der entsprechenden Programmstartadresse (200_8) gestartet. Läuft das Programm auf Anhieb fehlerfrei, so kann es auf Lochstreifen oder Band entweder im ASCII-Format oder in assemblierter Form abgespeichert werden (mehrere Kopien!).

Da der erste Probelauf meistens nicht erfolgreich verläuft, d.h. das Programm läuft z.B. in einer endlosen Schleife oder es liefert die falschen Ergebnisse, muß der Benutzer mit Hilfe der „Debugging Routine" das Programm in kleinen Stücken austesten. Um dieses Austesten möglichst effektiv zu gestalten, sollte das Testprogramm (Debugging Routine) die folgenden Möglichkeiten anbieten:

- Das zu testende Programm sollte ganz oder teilweise ausgelistet werden können, um nachzuprüfen, ob es richtig geladen wurde, ob nach einem Probelauf Programmteile zerstört wurden oder ob die gewünschten Ergebnisse erzielt wurden.
- Ein bestimmter Programmbereich sollte mit einer Konstanten geladen werden können, z.B. alle vom Programm nicht benutzten Speicherstellen sollen die Befehle HALT oder NOP als Inhalt haben. (→ Erkennen von Programmsprüngen außerhalb des im Programm angegebenen Bereiches).
- Der Inhalt eines bestimmten Speicherplatzes sollte geändert werden können.
- Der Programmbereich soll nach einem vorher festgelegten Speicherinhalt durchsucht werden, und dieser sollte mit Angabe der Adresse ausgegeben werden können.
- Bei Mikroprozessoren ist es oft vorteilhaft, wenn die Möglichkeit besteht, Registerinhalte (Akkumulator, Programmzähler, Statusregister) auszudrucken, um sie nachzuprüfen und gegebenenfalls zu verändern.
- Einfügen künstlicher Haltepunkte, sogenannter „Breakpoints". Das Einfügen eines „Breakpoint" hat zur Folge, daß das Programm an dieser Adresse anhält, sofern es diese Adresse überhaupt erreicht, und der augenblickliche Status, d.h. die Register der CPU ausgedruckt werden. Damit hat der Benutzer die Möglichkeit, sein Programm in kleinere Abschnitte zu zerlegen, deren Richtigkeit anhand der Registerinhalte geprüft werden kann. Ist der Programmabschnitt fehlerfrei, so kann das Programm an der unterbrochenen Stelle neu gestartet werden, um an einem weiteren Haltepunkt von neuem angehalten zu werden.

  Die Möglichkeit, „Breakpoints" einzubauen, versetzt den Benutzer in die Lage, sein Programm wirkungsvoll auszutesten, da er es in kleine, überschaubare Programmmodule zerlegen kann.
- Das zu testende Programm sollte nach einem HALT vom Testprogramm aus neu gestartet werden können; ebenso, wenn es sich in einer endlosen Schleife befindet.
- Das nach erfolgter Korrektur fehlerfreie Programm sollte an eine Ausgabeeinheit ausgegeben werden können. Es empfiehlt sich, die durchgeführten Korrekturen in die Programmliste einzutragen und das Programm mit dem Editor auszubessern und anschließend neu zu assemblieren.

Phase der Programmerstellung	Zur Verfügung stehendes Gerät	Verwendete Software	Ergebnis
Programmtestphase			
Test des assemblierten Quellenprogramms (object program)	RAM-Speicher des Entwicklungssystems (Test von Interrupt und I/O *möglich*)	Ladeprogramm und Debugging Routine des Entwicklungssystems	Ausgetestetes Programm in Maschinencode (kann in PROM, RAM-Speicher geladen werden)

Bild 6.11: Einsatzbereich der Debugging Routine

Zusammenfassung

Mit dem Editor, dem Assembler und der „Debugging Routine" ist der Programmierer in der Lage, seine Programme ausgehend von einem Problemprogramm [illegible] len, zu korrigieren, zu assemblieren und auszutesten. Die [illegible] Programmerstellung müssen je nach Fehlertyp wiederholt [illegible] ist bei einem Mikroprozessor vom Gesichtspunkt der [illegible] daß diese drei Systemprogramme mit den [illegible] genschaften zur Verfügung stehen, denn [illegible] beträchtlich.

Kapitel 7
Mikroprogrammierung: Definition und Grundlagen

7.1 Grundlagen

Der Zweck dieses Kapitels besteht darin, den Begriff „Mikroprogrammierung" zu erläutern und die grundlegenden Unterschiede in bezug auf die Architektur klarzumachen, die zwischen mikroprogrammierbaren Mikroprozessoren und solchen mit festverdrahtetem Steuerwerk bestehen. Dies soll im weiteren anhand von Bildern geschehen.

In Bild 7.1 und 7.2 sind die beiden verschiedenen Prinzipien zur Realisierung eines Steuerwerks dargestellt. Aus diesen beiden Bildern läßt sich der folgende Unterschied deutlich erkennen:

Bei mikroprogrammierbaren Mikroprozessoren sind Steuerwerk und Register-Arithmetikeinheit getrennt. Das Steuerwerk stellt, wie später noch gezeigt wird, einen selbständigen Prozessor innerhalb des Mikroprozessors dar. Bei herkömmlichen Mikroprozessoren, d.h. solchen mit festverdrahtetem Steuerwerk, bilden Steuerwerk und Register-Arithmetikeinheit ein Ganzes, die Zentraleinheit. *Mikroprogrammierung* hat es in erster Linie mit der *Realisierung des Steuerwerks* eines Mikroprozessors durch den *Anwender* zu tun. An dieser Stelle sei darauf hingewiesen, daß unter Mikroprogrammierung, wie des öfteren fälschlicherweise angenommen wird, nicht die Programmierung von Mikroprozessoren verstanden wird. Ehe auf die Grundlagen der Mikroprogrammierung eingegangen wird, sollen noch einmal die Aufgaben des Steuerwerks stichwortartig aufgeführt werden. Das Steuerwerk hatte folgende Aufgaben zu erfüllen:

- Bereitstellung eines neuen Befehls (Fetch-Zyklus).
- Dekodierung eines eingelesenen Befehls und dessen Zerlegung in eine Folge serieller und paralleler Anweisungen unter Zuhilfenahme eines externen Taktes.
- Überwachung und Steuerung aller arithmetischer und logischer Befehle der Register-Arithmetikeinheit.
- Überwachung der Ein- und Ausgabeoperationen.

Dem Steuerwerk fällt somit die Überwachung und Steuerung aller internen und externen Operationen zu, deren der Mikroprozessor fähig ist.

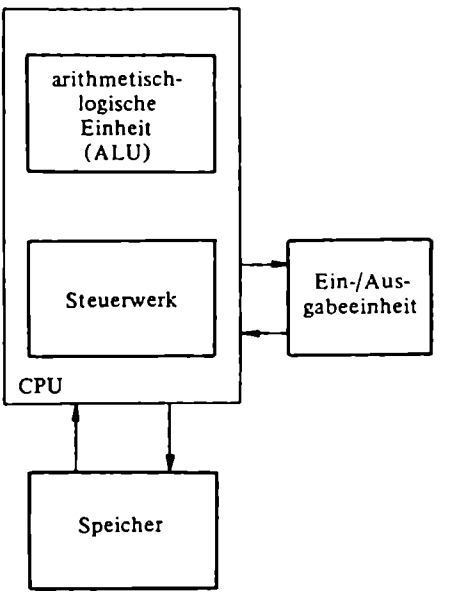


Bild 7.1: Konventionelles Mikroprozessorsystem

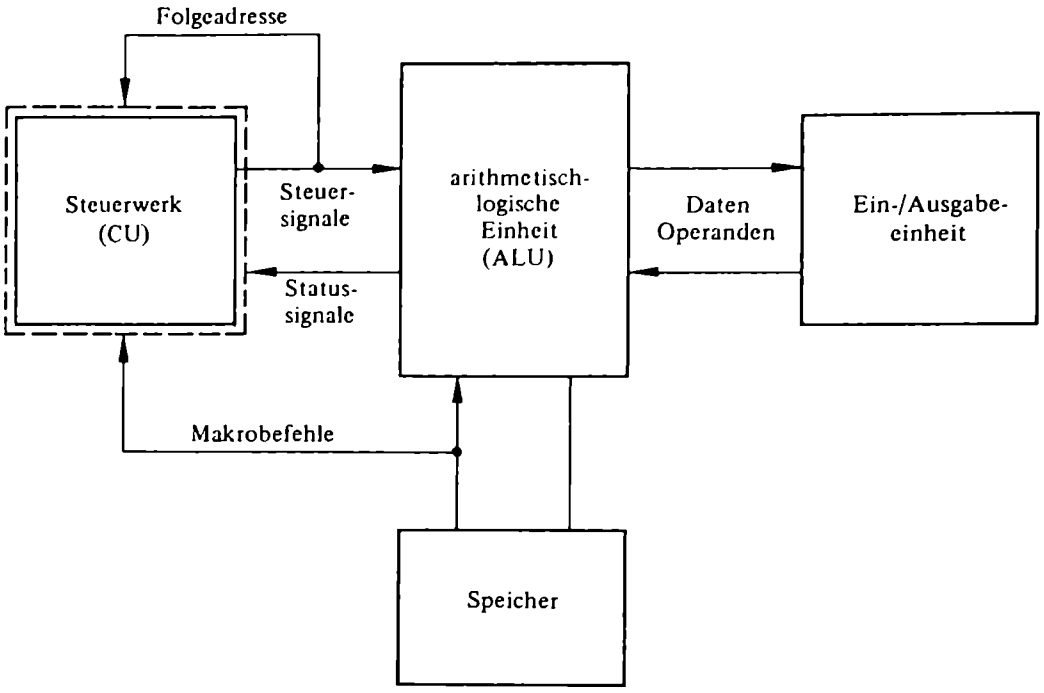
7.2 Gegenüberstellung: Herkömmlicher Mikroprozessor und mikroprogrammierbarer Mikroprozessor

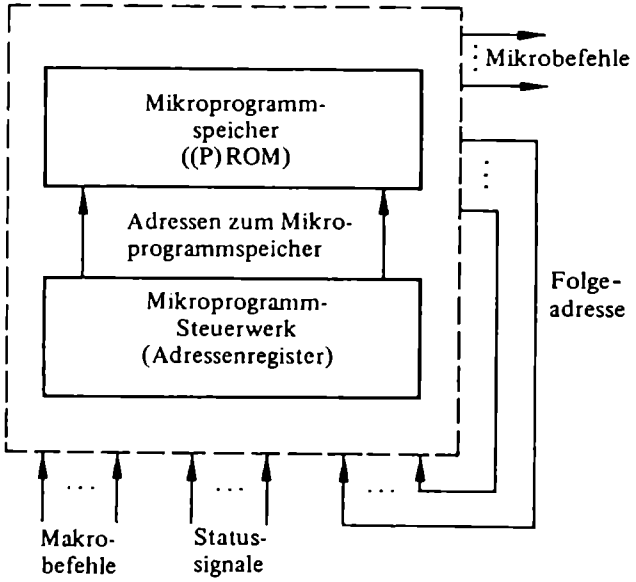
7.2.1 Herkömmlicher Mikroprozessor

Bei dem festverdrahteten Steuerwerk (Bild 7.1) werden die im Programmspeicher abgelegten Befehle eingelesen, indem der Programmspeicher adressiert wird (Fetch-Zyklus). Die Abarbeitung der Befehle erfolgt nach einem vom Mikroprozessorhersteller festgelegten Plan, der in Form von Zeitdiagrammen und Funktionsabläufen dem Benutzer zur Verfügung steht. Dabei ist festzuhalten, daß dem Benutzer nicht die Möglichkeit gegeben ist, verändernd in diesen internen Zeitplan des Mikroprozessors einzugreifen.

Über den Operationscode, der im Befehlsregister dekodiert wird, wird dem Steuerwerk mitgeteilt, welche Art von Befehl vorliegt (1-Byte, 2-Byte, 3-Byte), welche Adressierungsart gewählt wurde, d.h. in welcher Weise die im Adressenfeld angegebene Zahl zur Bildung der effektiven Adresse herangezogen wird und wohin das Ergebnis abgespeichert werden soll.

Alle diese Operationen werden unter alleiniger Regie des Steuerwerks und eines Taktes eingeleitet und ausgeführt. Der Benutzer hat *keine* Möglichkeit in den Ablauf dieser Operationen einzugreifen. In dieser Feststellung liegt der Hauptunterschied zwischen einem Mikroprozessor mit festverdrahtetem Steuerwerk und einem mikroprogrammierbaren begründet, bei dem über die Realisierung des Steuerwerks die Möglichkeit gegeben ist, in den internen Funktionsablauf einzugreifen und somit die nacheinander auszuführenden Operationen selbst zu gestalten. Dies soll im folgenden näher betrachtet werden.





Bestandteile des programmierbaren Steuerwerks

Bild 7.2: Mikroprogrammierbarer Mikroprozessor (ALU + CU)

7.2.2 Mikroprogrammierbarer Mikroprozessor

Definitionen

Im vorhergehenden Abschnitt wurde festgestellt, daß bei mikroprogrammierbaren Mikroprozessoren die Möglichkeit besteht, den internen Funktionsablauf selbst zu bestimmen. Dieser Vorgang wird Mikroprogrammierung genannt. Die wichtigsten bei der Mikroprogrammierung verwendeten Begriffe sollen anhand von Beispielen erläutert werden.

Der auszuführende Befehl sei ein *Additionsbefehl*, der im Programmspeicher abgelegt ist.

Dieser Befehl wird als *Makrobefehl* bezeichnet. Das vom Benutzer geschriebene Programm ist somit eine Folge von Makrobefehlen. Um die Verarbeitung der Makrobefehle zu klären, muß zunächst der Aufbau des Steuerwerks bei mikroprogrammierbaren Mikroprozessoren erläutert werden.

In Bild 7.2 ist ein derartiges Steuerwerk dargestellt. Es besteht aus einem schnellen Speicher, entweder ROM oder PROM, und aus einer Nächstadressenlogik. Von dem festverdrahteten Steuerwerk her ist bekannt, daß abhängig vom jeweiligen Befehl bestimmte für diesen kennzeichnende Operationen durchgeführt werden. Derartige Operationen sind zum Beispiel:

- lade Inhalt Akkumulator A in Schieberegister SR,
- lösche Akkumulator B,
- lade Akkumulator A ins Ausgaberegister,
- erhöhe Programmzähler um Eins,
- addiere Inhalt von Akkumulator A und Akkumulator B.

Ein vorgegebener Befehl wie der Makrobefehl „Addiere" wird in mehrere solcher Operationen, sogenannte Mikrobefehle zerlegt. Oder anders ausgedrückt: Mittels einer *Folge von Mikrobefehlen* läßt sich ein beliebiger *Makrobefehl* herstellen. Dies ist eines der bedeutendsten Merkmale der Mikroprogrammierung, denn sie erlaubt die Erstellung vom Anwender gewünschter Makrobefehle aufgrund einer Aneinanderreihung spezifischer Mikrobefehle. Somit kann sich der Benutzer seinen eigenen Befehlsvorrat (= Summe der Makrobefehle) schreiben.

Die Mikrobefehle können auch als Mosaiksteine betrachtet werden, deren sinnvolle Zusammensetzung zu einem Mikroprogramm im ROM- oder PROM-Speicher des Steuerwerks die Ausführung der im Befehlsvorrat vorgegebenen Befehle ermöglicht.

Die Zusammensetzung der Mikrobefehle zu einem Mikroprogramm stellt sehr hohe Anforderungen an den Anwender, und zwar in zweierlei Hinsicht:

- Hardware-Anforderungen,
- Software-Anforderungen.

7.3 Anforderungen an den Anwender

7.3.1 Hardware-Anforderungen

Jedem Mikrobefehl obliegen zwei Aufgaben, nämlich die Steuerung der Register-Arithmetik-Funktionen und zum anderen die Festlegung oder Mitbestimmung der Adresse des nächsten Mikrobefehls. Er besitzt wie jeder Makrobefehl ein bestimmtes Format. Dieses Format, dessen minimale Wortlänge meistens durch die Architektur und die zur Verfügung stehenden Mikrooperationen in der Register-Artithmetikeinheit vorgegeben ist, kann vom Benutzer festgelegt werden, d.h. es kann über die minimale Wortlänge hinaus ausgedehnt werden, um anwenderspezifische Funktionen zu realisieren. Ein solches Mikrobefehlsformat ist in Bild 7.3 dargestellt:

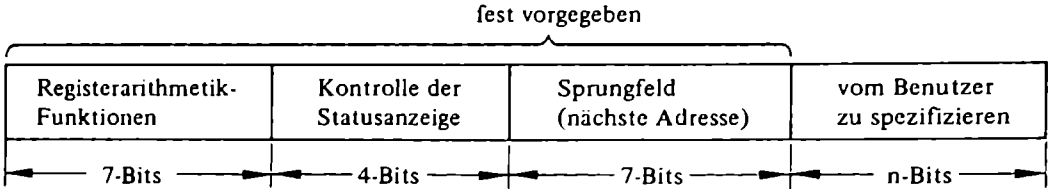


Bild 7.3: Mikrobefehlsformat

Ein Makrobefehl kann, wie bereits erwähnt, aus einer Folge von Mikrobefehlen aufgebaut werden. Dieser Sachverhalt zeigt, daß die Abarbeitung eines Makrobefehls voraussetzt, daß, abhängig von einem Takt, Mikrobefehle nacheinander abgearbeitet werden müssen. Jede Mikrooperation steuert während einer Taktperiode (ein Mikrozyklus) über einen Teil ihrer Bits eine Funktion der Register-Arithmetikeinheit (= Elementarfunktion) und ein anderer Teil wird zur Bestimmung der Adresse des nächsten Mikrobefehls verwendet. Ein bedeutender Teil des Mikrobefehlsformats ist der Bestimmung der Adresse des nächsten Mikrobefehls vorbehalten. Es werde angenommen, daß eine gerade auszuführende Mikrooperation aus dem Mikroprogrammspeicher ausgelesen wurde. Ein Teil der Bits kontrolliert die Register-Arithmetikeinheit und, falls vorgesehen, externe Bausteine (z.B. Programmspeicher, Ein-Ausgabebausteine), während der andere Teil den augenblicklichen Status der Register-Arithmetikeinheit abprüft und gegebenenfalls abspeichert sowie die Adresse der neuen Mikrooperation mitbestimmt, d.h. abhängig von der im Mikrobefehl eingestellten Sprungadresse sowie vom abgespeicherten Status der Register-Arithmetikeinheit (Sprungbedingungen) oder vom Operationscode des Makrobefehls wird ein neuer Mikrobefehl ausgewählt.

Daraus wird deutlich, daß Mikroprogrammierung vom Gesichtspunkt der Hardware aus an den Anwender folgende Anforderungen stellt:

- genaueste Kenntnisse über den Aufbau der verwendeten Register-Arithmetikeinheit (Architektur) (→ Kapitel 2!);
- genaueste Kenntnisse über den zeitlichen Ablauf der Funktionen in der Register-Arithmetikeinheit;
- genaueste Kenntnisse der zum Aufbau des Systems zur Verfügung stehenden Bauelemente;

- genaueste Kenntnisse der Systemarchitektur des zu realisierenden Systems;
- genaueste Kenntnisse des Zeitverhaltens des Gesamtsystems (→ Zeitverzögerung beim Adressieren des Mikroprogrammspeichers!);
- genaueste Kenntnisse über die Funktionsweise eines Mikroprozessors;
- genaueste Kenntnisse der internen Funktionsabläufe eines Steuerwerks;
- Abschätzung des Mikrobefehlsformats, abhängig von den Erfordernissen des zu realisierenden Gesamtsystems.

Diese Punkte machen deutlich, daß es sich bei Mikroprogrammierung um die Steuerung des internen Funktionsablaufs eines Mikroprozessors auf der Stufe der einzelnen Systembausteine handelt, zu denen der Anwender mit den einzelnen Mikrooperationen Zugriff hat und deren sinnvolle Hintereinanderschaltung die Realisierung eines mikroprogrammierbaren Systems ermöglicht (Bild 7.2).

7.3.2 Software-Anforderungen

Genaugenommen muß der Benutzer eines mikroprogrammierbaren Mikroprozessors zweimal ein Programm erstellen, und zwar zum einen das Mikroprogramm für das Steuerwerk und zum anderen das für seine spezifische Anwendung gedachte Programm, das Makroprogramm. Der Benutzer programmiert daher in zwei verschiedenen Ebenen. Soll ein vollkommen neuer Befehlsvorrat (= Summe von Makrobefehlen) definiert werden, so muß nach festgelegter Hardwarekonfiguration (Systemarchitektur) der für die spezifische Anforderung passende Befehlsvorrat ausgedacht und aufgestellt werden. Dies erfordert genaue Überlegungen über die Art der zu verwirklichenden Befehle, z.B. ob arithmetische, logische oder besondere Testbefehle und bedingte Sprünge nützlich sind oder ob auf wirkungsvolle Ein-/Ausgabebefehle besonderer Wert gelegt werden soll. Weiterhin muß festgestellt werden, ob Doppelgenauigkeitsbefehle und spezielle Befehle für Multiplikation und Division benötigt werden. Im weiteren müssen die Adressierungsarten, die bekanntermaßen die Verarbeitungsgeschwindigkeit eines Programms beeinflussen, definiert werden. Sind beide Prozesse abgeschlossen, so erfolgt die Zuordnung der entsprechenden Bits des Makrobefehls für die verschiedenen Operationscodes (Op-Code) und die Mode-Bits bzw. das Adressenfeld für die definierten Adressierungsarten.

Der nächste Schritt besteht in der Erstellung des Mikroprogramms, d.h. die Zerlegung der Makrobefehle in Elementarfunktionen (Mikrooperationen) und deren Anordnung im Mikroprogrammspeicher. Damit ist die Aufgabe des Steuerwerks verwirklicht, die die Funktionsabläufe innerhalb der Register-Arithmetikeinheit steuert. Zusätzlich müssen jedoch noch Funktionen verwirklicht werden, die, gesteuert durch ein Mikroprogramm, dafür sorgen, daß ein neuer Makrobefehl eingelesen wird („Fetch-Zyklus"), daß abhängig von der gewählten Adressierungsart ein Operand im Programmspeicher adressiert wird und der aufgebaute Mikroprozessor vor Programmbeginn initialisiert werden kann. Weitere Mikroprogramme müssen vorhanden sein, um dem Mikroprozessor die Reaktion auf externe Ereignisse (Interrupt-, DMA-Request, Power-Fail = Stromausfall) zu ermöglichen.

Zusammenfassend kann gesagt werden, daß sich vom Gesichtspunkt der Software folgende Anforderungen an den Anwender bei der Mikroprogrammerstellung ergeben:

- genaue Definition des zu realisierenden Befehlsvorrats (= Summe von Makrobefehlen);
- genaue Kenntnisse über Adressierungsarten und deren Auswirkung auf eine spezielle Anwendung;
- Zerlegung der Makrobefehle in eine Folge möglichst weniger Mikrobefehle, um schnell zu sein, d.h. einen Makrobefehl in möglichst wenigen Mikrozyklen auszuführen;
- Festlegen des Mikroprogramms im Mikroprogrammspeicher, wenn möglich unter Zuhilfenahme eines Mikroassemblers, der das Mikroprogramm im Speicher anordnet. (→ Begriff: Ein *mikroprogrammierter Mikroprozessor* besteht aus einem *internen Mikrocomputer*. Das Mikroprogramm übernimmt die Aufgabe des Steuerwerks, d.h. mit seinen über die Mikrobefehle auszuführenden Operationen steuert es den aufgebauten Mikroprozessor.)

7.4 Vorteile und Nachteile der Mikroprogrammierung

7.4.1 Vorteile

- Der Benutzer hat die Möglichkeit, seinen eigenen, auf seine Anwendung zugeschnittenen Befehlsvorrat zu entwickeln.
- Der Anwender ist frei in seiner Systemarchitektur, er kann die Wortlänge der Makro- und Mikrobefehle selbst wählen.
- Ein mikroprogrammierbarer Mikroprozessor ist vielseitig einsetzbar. Er kann z.B. mit einem Standard-Befehlsvorrat angeboten und beliebig je nach Anwendung erweitert bzw. abgeändert werden (zusätzlicher Mikroprogrammspeicher).
- Ein mikroprogrammierbarer Mikroprozessor ist meistens in bipolarer Technologie aufgebaut und daher schneller als die herkömmlichen Mikroprozessoren, wenn eine optimale Anordnung des Mikroprogramms vorausgesetzt wird.
- Da das Mikroprogramm und die Nächstadressenlogik das Steuerwerk bilden, ist eine bessere Überwachung der Funktionen der Register-Arithmetikeinheit und der externen Bausteine möglich, d.h. die Kontrolle der Funktionsabläufe erfolgt auf der Ebene der Systembausteine.

7.4.2 Nachteile

- An den Benutzer werden hohe Anforderungen bezüglich der Kenntnisse eines Mikroprozessors (Struktur, Funktionsablauf, Definition eines Befehlsvorrats) gestellt (→ schwierige Mikroprogrammerstellung).
- Für den selbsterstellten Befehlsvorrat gibt es keinen Assembler. Es besteht die Möglichkeit, über einen Makroassembler das Anwenderprogramm zu assemblieren.
- Ein Austausch von Makroprogrammen ist nur schwer möglich, da eine andere Anwendung auch wiederum ein anderes Mikroprogramm verwendet.

- Der Benutzer muß die verwendeten Bausteine (Register-Arithmetikeinheit, Ein-/Ausgabebausteine, Nächstadressenlogik, Mikroprogrammspeicher) in ihrem Zeitverhalten, ihren Funktionsmöglichkeiten und von der Architektur (Registeranzahl, interne Datenwege, Anordnung der Register, extern zugängliche Bussysteme) her genauestens kennen.
- Große Erfahrung ist notwendig, um ein Mikroprogramm zu erstellen (sonst sehr *zeitraubend*!).
- Es müssen zwei Programme erstellt werden (Mikroprogramm (Mikrobefehle) und Anwenderprogramm (Makrobefehle)).

7.4.3 Abschließende Bewertung

Mikroprogrammierung wird überall dort angewandt werden, wo ein Mikroprozessor benötigt wird, der speziell auf eine besondere Anwendung zugeschnitten sein muß, wo erfahrene Entwicklungsingenieure eine Mikroprogrammerstellung in angemessener Zeit garantieren und dort, wo es darauf ankommt, einen bereits vorhandenen herkömmlichen Mikroprozessor mit einem mikroprogrammierbaren nachzubauen, um auf diese Weise eine Verbesserung der Geschwindigkeit zu erreichen (→ Mikroprogrammerstellung!). Es muß aber besonders darauf hingewiesen werden, daß Mikroprogrammierung sehr hohe Anforderungen an den Systementwickler (Entwurf der Systemkonfiguration) und den Entwickler des Mikroprogramms bzw. Befehlsvorrats stellt. Wichtig ist vor allem eine sehr enge Zusammenarbeit des Systementwicklers und des Mikroprogrammentwicklers, um den Informationsaustausch in jeder Phase der Mikroprogrammerstellung und des Aufbaus eines mikroprogrammierbaren Mikroprozessors zu gewährleisten.

Literaturverzeichnis

[1] Digital Equipment: Introduction to programming. Maynard: Massachusetts, Volume 1 (1972)

[2] Korn, G.A.: Minicomputers for Engineers and Scientists. New York: McGraw-Hill, 1973

[3] Souček, B.: Microprocessors and Microcomputers. New York: John Wiley and Sons, 1976

[4] McCluskey, E.J.; Gschwind, H.W.: Design of Digital Computers. Berlin: Springer-Verlag, 2. Auflage, 1975

Stichwortverzeichnis

Addition 18
  BCD 29
  duale 18
  oktale 26
  hexadezimale 26
Additionstechniken 37
Adressen
  symbolische – 172, 182 f.
Adressenberechnung 79, 80
Adressenfeld 80, 82
Adressenzwischenspeicher 70
Adressierungsarten 79 ff., 146
  absolute 96
  absolute zero page 96
  autodecrement 90, 127
  autoincrement 89, 127
  Beispiele 138 ff.
  direct 96
  direkte 85 ff., 99
  extended 96
  hypothetischer Mikroprozessor 116
  immediate 94
  implied 86
  indirect 106
  indirect autodecrement 103
  indirect autoincrement 102
  indirect relative 107
  indirekte 100 ff.
  inherent 86
  page relative 98
  postindexed indirect 104
  preindexed indirect 104
  Register 87
  Register direct 88
  Register indexed 92, 96, 98, 138
  Register indirect 92, 101
  Register indirect indexed 104
  Register relative 92
  relative 98
  Symbole 84
Adreßleitungen 66, 73, 79
Adreßregister 66, 83, 113, 124
Adreßteil 114 ff.
Akkumulator (ACC) 33, 44 f.
  Definition 45
Akkumulator-Arithmetikeinheit (ACC, ALU) 45
  Befehle 46 ff.
  Blockschaltbild 57
Anfangsadresse 183
Argument 130, 186
Argumententransfer 130, 133 ff.
arithmetic overflow 22
Arithmetikeinheit 41 ff.
  Blockschaltbild 41
arithmetisch-logische Einheit (ALU) 33
  Aufbau 35 ff., 45
  Befehlsvorrat 35
Assembler 172, 179 ff.
  Drei-Pass 182
  Einsatzbereich 186
  Fehlermeldungen 184 f.
  Leistungsmerkmale 183
  symbolischer – 179
  Zwei-Pass 182
Assembleranweisung 182
Assemblersprache 172, 176, 179
Assemblierung
  bedingte – 185
Assemblierungsphase 173, 186

Basisregister 82, 113
BCD-Zahlen 29, 112, 117
Befehl
  – Dekodierung 68
  – Verarbeitung 74 f.
  1-Byte- 81, 182
  2-Byte- 81, 182
  3-Byte- 81, 182
Befehle
  Arithmetische – 117 ff.
  Bitkontroll- 120

Eingabe/Ausgabe- 137, 154
Haltbefehl 109
Logische – 119 ff.
Register- 124
Sprung- 127
Test- 121
Transfer- 122
Unterroutine- 130
Vergleichs- 120
Verschiebe- 122
Verzweigungs- 127
Befehlscode 60
Befehlsformat 80 ff.
hypothetischer Mikroprozessor 114 f.
Befehlsregister 65 f., 75
Befehlsvorrat
Akkumulator – Arithmetikeinheit 46, 48 f.
Beispiele 138 ff.
hypothetischer Mikroprozessor 109 ff.
Register-Arithmetikeinheit 64
Schieberegister 63
Befehlszyklus 75 f.
Bestimmungsort 60
Bewertungsfaktor 14
Binärformat 116, 179
Bitmuster 60, 79, 116
Borrow 19
Breakpoint 188
buffer pointer 176
BUS 66
BUS-Struktur 149 f.
Adreß- 73, 150
Daten- 150
Einfach- 150
Mehrfach- 150
Zweirichtungs- 150
Byte 81

carry 18, 36
Carry-Flag 111, 122
Clockgenerator 67, 71
Code-Feld 181
Comment-Feld 181
computed addressing 92
conditional assembly 183
cycle stealing 169

Datenaustausch 70, 76
bedingter – 156
-Beispiel 157
programmierter – 155
unbedingter – 155
-verfahren 149 ff.
Datenformat 79 f., 150
Datenleitungen 66, 73 f., 79
Datenregister 66, 70, 75
Debugging Routine 173, 187
Einsatzbereich 188
Leistungsmerkmale 188
Defer-Zyklus 75
Dekodierung 68, 79
Device-Flag 156
Dezimal-BCD-Umwandlung 28
Dezimal-Dualumwandlung 15
Dezimalkorrektur 29, 117
Dezimalsystem 13
Dezimalzahl 13
Displacement 81 f., 98
Distanzadresse 81 f., 98
Division 22, 50, 53, 123
Divisionsmethode 16
DMA-Transfer 168 f.
Dual-Dezimalumwandlung 15
Dualsystem 14
Dualzahl 14

Editor 174 ff.
Arbeitsweise 175
Befehle 177 f.
Befehlszustand 175
Definition 171
Einsatzbereich 178
Textzustand 175
Editorphase 173, 178
Effektive Adresse (EA) 82
Einadreßbefehl 60, 114
Einerkomplement 20
Eingaberegister 33, 69 f.
Entwicklungssystem 171 f.
Execute-Zyklus 76

Festwert-Speicher 71
Fetch-Zyklus 69, 75
Flußdiagramm 138, 173
Folgeadresse 192

Gewichtung 14
Gleitkommadarstellung 30

Halbaddierer 35
Half-Carry-Flag 112
Handshake-Transfer 157 ff.
  Flußdiagramm 159
  Impulsdiagramm 158
Hardware-Stack 113
Hauptprogramm 130, 132
Hauptspeicher 109
Hexadezimalsystem 25
hypothetischer Mikroprozessor
  Adressierungsarten 116
  Befehlsvorrat 110 ff.

Indexregister 65, 92, 110, 124
Interface 149 ff.
  Anforderungen 150 f.
  Aufbau 152
  DMA- 168
  Register 153
Interrupt
  Bedienung 162, 165
  Identifikation 162
  nesting 163, 165
  request 162 f.
  Statuswort 163
  vectored – 167
interrupt acknowledge 162
Interrupt-Mask-Flag 112
Interruptsystem 112, 162 ff.

Klammersymbol 76
Korrekturfaktor 29, 117

Label-Feld 181
Last in first out (LIFO) 113
least singificant bit 14
Leitwerk 67 f.
Linksverschiebung 122
Listenverarbeitung 89, 146 ff.
Look-Ahead Carry 38, 40, 54 f.

Makro 186
Makrobefehle 193 f.
Maschinencode 172, 180
Mehrfachgenauigkeitsdarstellung 30
Mikrobefehle 193, 196
Mikrobefehlsformat 195
Mikrocomputer 33 f., 68 ff.
  Baugruppen 33 ff.
  Definition 33
Mikroprogrammierung 191 ff.
  Nachteile 197
  Vorteile 197
Mikroprogramm
  Hardware-Anforderungen 195
  Software-Anforderungen 196
Mikroprogrammspeicher 193
Mikroprogrammsteuerwerk 193
Mikroprozessor
  Definition 33
  herkömmlicher – 192
  mikroprogrammierbarer – 193 f.
Mikroprozessorsystem 33, 109
Mikrorechner 33
Mikrozyklus 63, 195
Mnemonic 61, 116, 172, 179
Mode-Bit 81 f.
Monitorprogramm 187
most significant bit 20
Multiplikation 18, 27, 50 ff., 123
Multiplikationsmethode 30
Multiplikationsunterprogramm 133 f., 144

Nächstadressenlogik 194
Negative-Flag 112
no operation 43

object code 182
object program 180
offset 82, 92, 98
Oktal-Dualumwandlung 24
Oktalsystem 23
Op-Code 79 ff.
Operand 45, 73, 79, 130
Operand-Feld 181
Operationscode 56, 60, 79
Overflow-Flag 22, 111, 123

page-Adresse 96
Peripherie 109
Peripheriegerät 121, 149 ff.
pointer address 100
Polling Routine 162, 166
Priorität 162
program debugging 187
Programmable read only memory 71
programme interrupt 157, 161, 168

Programmerstellung 171 ff.
  Definitionen 171
  Phasen 173
  Vereinbarungen 171
Programmliste 181, 183, 186 ff.
Programmstruktur 146
Programmtestphase 173
Programmunterbrechung 157, 161
Programmverzweigung 121, 127
Programmzähler 65, 67, 71, 75
Programmzähleradressierungsarten 83, 94 ff., 106 ff.
Pseudobefehle 185
Punkt (dot) 83
push-down list 113

Quellenprogramm 176, 183

Random access memory (RAM) 70
Read only memory (ROM) 70
Rechenelement 54 ff.
Rechtsverschiebung 122
Registeradressierungsarten 83, 86 ff., 101 ff., 113
Register-Arithmetikeinheit 58 ff., 191
  Befehlsvorrat 64
  Blockschaltbild 59
  Funktionsbeispiel 65
Registerbereich 67, 71, 79, 113
Reprogrammable read only memory (RePROM) 71
Ripple-Carry-Addierer 37 ff.
Rückkehradresse 131

Schieberegister 60, 63
Schreib-Lese-Speicher 70
Seite (page) 96
Sense-Leitung 156, 161
signed binary number 111, 123
signed shift 123
Softwarehilfsmittel 171 ff.
  Einsatzbereich 173
Sortierroutine 141 f.
source program 178, 180
Speicher 33, 70 f.
  Kapazität 70
  Organisation 70, 74
  Raum 73 f., 79 ff.
Sprungadresse 183
Sprungbedingung 128

Stack 113, 126, 131, 162
  Pointer- 126
  Software- 113
Stackarchitektur 133
Stack Pointer 65, 110, 113, 124
state 63
Statusregister 65, 71, 75, 110 ff.
Stellenwert 13 f.
Steuerleitungen 66, 79
Steuerwerk 33, 67 f.
  festverdrahtetes – 191
  internes Zustandsdiagramm 69
  mikroprogrammierbares – 191
Subroutine 130
  Anruf 131
  closed – 131
  linkage register 132 f.
  nesting 113
  open – 131
  reentrant 113
  Rücksprung 131
Subtraktion 19, 26
Subtraktionsmethode 15
symbolische Adressen 172
symbolische Befehle 172
symbol table 182
Symboltafel 182
Syntaxfehler 182
Systeme
  mit Interrupt 162
  ohne Interrupt 161

text buffer 175 f.
Transfer 151
  asynchroner – 151, 157
  bedingter – 156
  DMA- 168
  Handshake- 157 ff.
  synchroner – 155
  unbedingter – 155
True-Complement-Schaltkreis 41

Übertrag (carry) 18, 36
unsigned binary numbers 123
unsigned shift 123
Unterprogramm 130
Ursprungsort (source) 60

vectored interrupt 167
Verschiebebefehle 143

Verzweigungen
  bedingte – 127 ff.
  unbedingte – 127
Volladdierer 36
vorzeichenbehaftete Dualzahl
  Division 53
  Multiplikation 51 f.
Vorzeichenbit 20, 111

wahlfreier Zugriff 92

Zahlenbereichsüberschreitung 22
Zahlendarstellung 13
Zahlensysteme 13
Zeigeradresse **100**, 113
Zentraleinheit (CPU) 33
  busorientierte 65 ff.
Zero-Flag **111**
Zustände (states) 63
Zweiadreßbefehl **60**, 114
Zweierkomplement 20 f.
Zweierkomplementarithmetik 20
Zweierkomplementdarstellung 20
Zweirichtungsbus 150

Nach dem Studium der Nachrichtentechnik mit Schwerpunkt auf der Übertragungstechnik und der Datenverarbeitung ist der Verfasser heute Mitarbeiter der Standard Elektrik Lorenz AG in Stuttgart. Sein Arbeitsgebiet sind Untersuchungen an Mikroprozessoren.

Das Thema ist bisher in der deutschen Literatur wegen der überaus raschen technischen Entwicklung nur wenig behandelt. Hier liegt nun ein ausführliches Lehrbuch vor. Es befaßt sich im Gegensatz zu üblichen Firmendarstellungen nicht mit speziellen Bauelementen, sondern zeigt die Grundlagen des Aufbaues und der Funktion von Mikroprozessoren auf. Ihre besonderen Probleme, die durch die hochintegrierte Bauweise bedingt sind (z.B. begrenzte Zahl von Anschlußleitungen und begrenzte Befehlsvorräte) werden schwerpunktmäßig behandelt. Typisch ist hier die erhöhte Bedeutung der Adressierung, der ein besonderer Abschnitt gewidmet ist. Darüber hinaus wird der Leser eingeführt in die Technik des Verkehres mit der Außenwelt, d.h. mit Peripheriegeräten. Hier stehen besonders die Datenaustauschverfahren im Vordergrund. Die Programmierung der Mikroprozessoren und die damit zusammenhängenden software-Aspekte werden eingehend erörtert, da sie für den lernenden Studenten oder Ingenieur, die bisher mit dieser Technik wenig Berührung hatten, besonders wichtig sind. Der vorliegende Teil I schließt mit einer Darstellung der Mikroprogrammierungsmöglichkeiten.
Teil II wird Anwendungen und weitere Beispiele bringen.

Reihe Datenverarbeitung
R. Oldenbourg Verlag München